Tutorial 3

Basic Data Structures and Algorithms

THINGS TO LOOK FOR...

- Definition and uses of containers.
- Array and list based containers.
- Designing and building the linked list, queue, and stack data types.
- The utilization of interfaces to establish behavior.
- Variations on the linked list data structure.
- The need for searching and sorting.
- Design and implementation of the linear and binary search algorithms.
- Design and implementation of the selection sort and quicksort algorithms.

3.0 INTRODUCTION

In this tutorial, we will look at several applications of the C language that we will find useful in designing and developing embedded systems. These applications will comprise three fundamental data structures, the linked list, the queue, and the stack, and four algorithms, linear search, binary search, selection sort, and quicksort. Each of these should be a basic tool in every embedded developer’s tool box.

The data structures that we will develop fall into the general category of what we call containers. We use these to hold all different kinds of data both within a process and as shared variables between processes.

3.1 Array Based Containers

The array, which often underlies the data type we call a buffer, is a very convenient data structure to use as a container. It is efficient and supports fast, random access. For certain kinds of applications it works very well. However, arrays are not without problems. First, they are not particularly flexible. When their size must be changed, we must allocate new memory, copy the old array into the new location, and then delete the old array. This is generally not feasible in the firmware environment that we typically find in an embedded application. Arrays can also be the source of many hard to find errors if one does not carefully manage the boundaries of the structure when defining or accessing it.
3.2 Lists and List Based Containers

As one alternative to the array, we introduce a container called a list. In its generic sense, the list is something that we probably use every day. On reflection, we can ask, what exactly is a list? We can begin with a list of names, groceries, numbers, girlfriends, boyfriends, cousins, nieces, Sherpa guides to the Himalayas…even this collection of things. More formally, we see, then, that a list is a finite sequence of elements. We recognize that a list may be empty. The list of all my wealthy and famous friends, for example.

Up to this point, we’ve been simply naming collections of things. We’ve said nothing about the structure of the list or the order of its elements. Let’s think about building a list…in an abstract sense. What features and characteristics do we expect to see? These are our use cases; our outside view.

- Create list
- Insert item
- Remove item
- Change item
- View item
- Check size

These are given in the Use Case diagram in Figure 3.0. We’ll cover the textual description and exceptions for each of the use cases when we present the design.

If we think about the list first as something that we might be writing on a scrap of paper or in a notebook, we can see that it has some interesting properties. A list is a sequence that supports random access to its members. We can select any element from the list at any time. We should be familiar with the term from working with simple arrays.

The amount of time that it takes to add or to remove something from the end does not depend upon how many elements there are in the list. We just add the new element on or delete the old. We call this constant time access. Such a capability is potentially quite useful in an embedded application where time constraints are a routine part of a specification.

Constant time insertion and removal of elements at the end of an array means that elements of the array don’t have to be moved. When the term is applied to a list we mean that the list does not have to be rewritten to add new elements at the end. Contrast this with adding elements either at the beginning or in the middle of the list. Such operations may entail moving some or all of the contained elements as we see in Figure 3.1.

We also see that the time it takes to insert or remove something from the middle or the start of the list gets longer as we add more elements – that’s a lot of erasing and recopying. We call this linear time access. This simply means that the time to perform the task gets longer as a linear function of the number of elements in the container.
The number of elements in the list may be dynamic; the list should be able to grow without intervention. We should be able to determine the list’s size – the number of valid elements it contains. We contrast size with the list’s capacity, the number of elements that the space we’ve set aside for it will hold. Both of these pieces of information are necessary. The distinction becomes important when inserting element into list for which size and capacity are equal. Under such circumstances, the capacity of the list must be increased. Remember that the access time will increase linearly with size.

3.3 Linked Lists

We can see that a list type container offers a powerful alternative to the array. A data structure that can begin to give the capabilities that we’re looking for is called a linked list. The linked list is structured much like a chain. If we want to make the chain longer, we add a link. If we want it shorter, we remove a link. It’s easy to add or to remove links at either end of the chain; we can add or remove elements of linked list in a similar way. For the chain and the linked list, adding or removing elements in middle of the structure is more difficult.

3.3.1 Designing a Linked List

As we examine the linked list in greater detail, we will also be illustrating some of the thought process that one goes through in executing the design of any software or hardware module. As a way to manage the complexity of a new design, we begin by looking for related concepts that we may already understand. While the linked list is not a particularly complex design, the steps we use remain appropriate for those designs that are.

In the opening discussion on lists, we've identified some of the desirable high level requirements. Let's now begin to formalize the design. The first steps involve taking a look at the design from the outside, that is, we take an external view. Such a view reflects how our clients will use our system. We then move to the inside as we begin to develop the system that will give rise to the required features and capabilities. We start by specifying the interface and then proceed to the implementation. We first ask what then follow with how. The use cases for the list that we identified earlier give the first cut at our external interface - the what part.

Defining the Interface

We begin outside of the list. We need to ask what behaviors are required.

1. Identify and quantify the desired operations

This process begins with studying the application(s) that will use the list. We must talk with the potential users. We must do so in their language; the language of their application. From such discussions, we can establish the requirements (which can be expressed in a variety of ways). UML use case diagrams or pseudo code are often very effective tools to capture and express such information.

2. Map the abstract operations to access methods.

These are the methods that people will use to interact with the system. These are its public interface; they determine the its perceived behavior. The goal is to provide a clean, well defined, persistent interface that allows the client to access the internal data of the list in a convenient, controlled, and robust way.
Defining the Implementation

Next, we move inside of the list. We must think about how to implement the specified behaviors.

3. Decide on the implementation functions and the internal data representation.

These are the internal functions, variables, and their structure. At this point, we want to hide this implementation.

4. Once coded, the list now has the declaration and the definition or implementation portions. We put the declaration into a header file and the implementation into an implementation file, a .c or .lib file.

This all seems like a lot of work. Why don't we just tell the client to use an array, and be done with it? This is a good question. Let's look at just a few things first.

With the basic array, the client has direct access to the data. They can enter or change the contained data as they wish. With the linked list, visibility of the internal data is restricted to the access functions; we, the implementers, are in control. We can limit the data to a certain format or range of values, for example. When using the array, the client must manage the size and capacity of the array then allocate extra memory if necessary. With the linked list, the allocation can be taken care of automatically. With the linked list, we can apprise the client that they are trying to extract from an empty container or enter data in to a full one. With the basic array, the client must manage all of that themselves. We develop the linked list to provide the client with a richer and more robust set of tools than those offered by the intrinsic data structures.

In a larger sense, the primary goal of developing tools such as the linked list is to strive to continually improve the overall quality, robustness, and reliability of our designs. Maintaining a library of tools that have been well designed then extensively tested and widely used enables us to work with them in future designs with confidence.

3.3.1 Nodes

Returning to the earlier chain - linked list analogy, for a chain, the fundamental component is the link. The user is able to remove or to add links to shorten or lengthen a chain. For the linked list, that component is a node. Nodes can similarly be removed, added, or manipulated to modify or extend the linked list.

Nodes are also a simple container in their own right. They provide the means for storing data and a method for identifying the next element in the list. More formally, we see that a basic node comprises

- A field to contain a data item
- A reference to next element or node in list

Like a link in a chain, the reference connects each node (link) to the node (link) on its right. We can express the concept graphically as a UML object or class diagram given in Figure 3.2.

![Node Class Diagram](image)
3.3.1.2 The Linked List

Combining collections of nodes in an organized way gives a schematic picture of the evolving linked list concept in Figure 3.3. The first element or node is called the head. The last is called the tail. We say that the tail is grounded (NULL) to identify the end of the list. We can see that we can move through the linked list by following the next reference from one node to next. When we reach a node whose next reference is NULL, we know that we have reached the end of the list. When the head has a NULL value, we know that the list is empty, it contains no elements.

![High Level Model of a Linked List Data Type](image)

Formalizing the high level diagram, we see that the linked list is composed of a set of nodes as is given in the UML diagram in Figure 3.4. Observe that in our implementation, we have chosen composition rather than aggregation.

![Linked List as a Composition of Nodes](image)

3.3.1.3 Defining Operations

Based upon the earlier use case diagram, the minimal set of requirements the list should support are,

- Create a list – `createList()`
- Add an element – `addElement()`
- Delete an element – `deleteElement()`
- Modify an element – `modifyElement()`
- Retrieve an element – `getElement()`

Let’s first look at each from a high level perspective. We assume for the moment that error conditions are not being managed.

Creating a Linked List

Creating a linked list in an embedded context can be tricky. Generally we don’t have the luxury of dynamic memory allocation. The root of the list must be created at compile time; the Node instances must be created then as well. We hold them in a designated part of memory until we need them.
Adding or Deleting an Element

When we add or delete an element to/from the linked list, we must consider 3 cases

- Add/Delete at tail
- Add/Delete at head
- Add/Delete in middle

Let’s see how to do each of these.

Adding at the Tail

This one is rather straight forward. We simply follow the next references to the end of the list then change the last `next` reference to refer to the new node and set the new node `next` reference to point to NULL (unless it was initialized that way).

Adding at the Head

If the list is empty, then set the head reference to refer to the new node. Otherwise, set the new node’s `next` reference to the value of the head reference and set the head reference to refer to the new node.

Adding in the Middle

Start with the head reference and follow the next references to the position where the new node is to be added. Set the new node’s `next` reference to the value of the `next` reference of the current node. Then, set the current node’s `next` reference to refer to the new node.

Removing from the Tail

Follow the list to the end and change the next reference of the node preceding the last node to NULL.

Removing at the Head

If the list is empty, signal an error. If the list contains only a single element, set the `head` reference to NULL. Otherwise, set the value of the `head` reference to the value of the `next` reference in the first element.

Removing from the Middle

Follow the list to the position one before where the node is to be removed. Set the current node’s `next` reference to the value of the `next` reference of the following node.

Modifying an Element

Start with the head reference and follow the `next` references to the position of the specified node. Change the node’s value.

Retrieve an Element

Start with the `head` reference and follow the `next` references to the position of the specified node. Get the node’s value and return it.
3.3.2 Detailed Design and Building the List

We now have a high level model of the design. Let’s look at the next level of detail.

Our tasks comprise
1. Designing the Node object
2. Implementing the Node object
3. Designing the Linked List
4. Implementing the Linked List
5. Designing the access methods for the Linked List
6. Implementing the access methods for the Linked List
7. Testing the design

3.3.2.1 Designing the Node Object

We can design and implement a Node rather easily using a struct. The struct should have a minimum of two data members, one to hold the data and one to hold the next reference. It seems natural to select Node as the identifier for the struct. Thus, the reference becomes a pointer to a Node. We declare the Node as shown in Figure 3.5.

The declaration looks a bit unusual since the structure type has a member that is a pointer of its own type. The first portion of the first line of the body, struct node, is called an incomplete or partial declaration. Such a declaration is allowed in the language to permit a structure type to have a pointer to itself or to another instance of the same type. Without such a pointer, building linked list types of containers would be significantly more difficult.

Like the elements in an array, the data held in each node in the linked list must be of the same type or combination of types. For the current example, we will specify a single data member of type integer.

3.3.2.2 Designing the Linked List

The design will support
- Inserting Node at the head, tail, or named location.
  The function will accept head pointer, location, and new Node
  Return void
- Deleting Node at the head, tail, or named location.
  The function will accept head pointer and location
  Return void
- Modifying data at the head, tail, or named location.
  The function will accept head pointer, location, and data value of type int
  Return void
- Returning data at the head, tail, or named location.
  The function will accept head pointer and location
  Return data value - type int
- Returning size
  Return number of elements - type int
The public interface to the linked list will support the following methods,
createList()
insertNode()
deleteNode()
size()
setValue()
getValue()

These are now given in the modified class diagram for the linked list presented in Figure 3.6.

cREATELIST() Method

The head of the list is created at compile time and initialized to NULL.

insertNode() Method

Assumptions

For simplicity, it will be assumed that the requested position will be valid. This assumption is not valid in practice and must be managed in the design.

Algorithm

The algorithm for inserting a node into a linked list is given in Figure 3.7.

if the list is empty
    if the position is 0
        set as the head
    else
        if the position is 0
            insert at the head
        else if
            at the end of the list and the position is 1
            add the element
        else
            while the list is not empty
                find the position
                insert the element
        end if
end if

deleteNode() Method

Assumptions

The requested position will be valid. This assumption is not valid in practice and must be managed.
Algorithm

The algorithm for deleting a node from a linked list is given in Figure 3.8.

```
if position == 0 and size == 1
    head ← null
else if position == 0
    head ← next of node to delete
else
    find node at (position -1)
    node next ← next of node to delete
end if
```

Figure 3.8
Delete Node

size( ) Method

Assumptions
List not empty. This assumption is not valid in practice and must be managed.

Algorithm
The size of the linked list is returned using the simple method in Figure 3.9.

```
return size
```

Figure 3.9
size Method

The setValue and getValue methods are presented in Figure 3.10 and Figure 3.11 respectively.

setValue( ) Method

Assumptions
The requested position will be valid and the new data will be of the correct type. These assumptions are not valid in practice and must be managed.

Algorithm
The method for altering the data value stored in a node is given in Figure 3.10 below.

```
find node at position
node data ← new data
```

Figure 3.10
setValue Method

getValue() Method

Assumptions
The requested position will be valid. This assumption is not valid in practice and must be managed.
Algorithm

The method for retrieving the data value stored in a node is given in Figure 3.11.

```
find node at position
return node data
```

Figure 3.11

getValue Method

3.3.2.3 Implementing the Linked List Class

The listing in Figure 3.12 creates the linked list and implements the insertNode() method and a showList() method to test the operations. The remaining methods are left as exercises.

```
#include <stdio.h>
// declare the underlying data structure and access function prototypes
typedef struct node
{
    int myData;
    struct node* next;
} Node;

// function prototypes
void insertNode(Node** headPtr, Node* nodeToAddPtr, int position);
void showList(Node* headPtr);

void main(void)
{
    // declare and initialize some test nodes
    // and pointer to the head of the list

    Node node0 = {0, NULL};
    Node node1 = {1, NULL};
    Node node2 = {2, NULL};
    Node node3 = {3, NULL};
    Node* headPtr = NULL;

    // build the linked list and test it
    insertNode(&headPtr, &node0, 0);
    showList(headPtr);
    insertNode(&headPtr, &node1, 1);
    showList(headPtr);
    insertNode(&headPtr, &node2, 1);
    showList(headPtr);
    insertNode(&headPtr, &node3, 0);
    showList(headPtr);
    return;
}
```

Figure 3.12

Implementing a Linked List
void insertNode(Node** headPtr, Node* nodeToAddPtr, int position)
{
    Node* tempPtr = *headPtr;
    if (tempPtr == NULL) // if the list is empty
    {
        if (position == 0)
        {
            *headPtr = nodeToAddPtr; // insert at the head
        }
    }
    else
    {
        if (position == 0)
        {
            nodeToAddPtr->next = tempPtr; // insert at the head
            *headPtr = nodeToAddPtr;
        }
        // at the end of the list and the position is 1
        else if ((tempPtr->next == NULL) && (position == 1))
        {
            tempPtr->next = nodeToAddPtr; // add the element
        }
        else
        {
            // find the position
            while((--position !=0) && (tempPtr->next != NULL))
            {
                tempPtr = tempPtr->next;
            }
            // insert the element
            if (position == 0)
            {
                if (tempPtr->next == NULL)
                {
                    tempPtr->next = nodeToAddPtr;
                }
                else
                {
                    nodeToAddPtr->next = tempPtr->next;
                    tempPtr->next = nodeToAddPtr;
                }
            }
        }
    }
    return;
}

Figure 3.12 cont.

Implementing a Linked List
3.3.3 Common Errors

Some common errors that may occur during the design of the linked list include,
• Failure to initialize a pointer or node.
• De-referencing a null pointer.
• De-referencing an incorrect pointer.
• Going beyond the end of the data structure.

3.3.4 Some Variations on the List ADT

There are a number of simple variations on the basic linked list that are quite useful when designing embedded applications.

Doubly Linked Lists

Doubly linked lists add a second pointer to the Node definition to allow one to link to a previous node as well as to the next. The simple addition makes traversing the list in either direction rather easy. Although the design may consume a little more space and add some complexity for managing the extra pointers, sometimes it’s worth it. Here, we are trading off complexity in the design for ease of use and potentially improved temporal performance at runtime.

Circular Lists

Circular lists are very powerful structures in certain applications. Such data types can remove some boundary cases of pointer management that we’ve encountered (although they may create others). We implement a circular list by simply setting the next pointer of the last node to the address of the first node rather than NULL.

Head and Tail Pointers

These are particularly useful when we implement queues as well. We always add at the tail and remove from the head.

```
/*
 * print the list
 */
void showList(Node* headPtr)
{
    Node* tempPtr = headPtr;
    if (tempPtr != NULL)
    {
        while( tempPtr->next !=NULL)
        {
            printf("%d \n", tempPtr->myData);
            tempPtr = tempPtr->next;
        }
        printf("%d \n", tempPtr->myData);
    }
    printf("\n");
    return;
}
```
3.4 Queues

We should already be familiar with the term *queue*... we run into enough of them almost daily. We queue at the bank, we line up or queue for traffic, we do the same for the ferry, and for lunch, we use one for playing pool...oops, wrong type. What do we see as the common behavior for a queue? If people are behaving properly, then they will only enter the queue at the back and they will only leave from the front. In contrast to the list which supports random access, we say that the behavior of the queue is *first-in, first-out*. In embedded applications, as in real life, cutting into the middle of a queue is frowned upon.

As with the other data types, such real life situations should suggest the design of the queue as well as potential applications in modeling, simulation, and managing data flow in embedded systems. Such access restrictions should also suggest how the access time is affected by the number of elements in the queue. Formalizing the model, one can state that a *queue* is a data structure of objects with the constraint that objects can only be inserted at one end ... called the *rear* or *tail* and removed at the other end...called the *front* or *head*. The object at the front end is called the *first entry*.

The front is the first element in the queue and the rear is the last element as we see in the model in the diagram in Figure 3.13. The behavior is called *FIFO First In First Out*.

![Queue Diagram](image)

Figure 3.13. High Level Model for a Queue Data Type

3.4.1 Abstract Queue Operations

As we did when designing the list, we start the process by thinking about the functionality or operations that must be supported. As we’ve stressed, these operations are our public interface; they define the behavior of our system. The use case diagram for the queue data type is given in Figure 3.14.

![Use Case Diagram](image)

Figure 3.14. Use Case Diagram Queue Data Type

3.4.1.1 Defining Operations

From the use case diagram, we determine that the minimal set of requirements that the queue should support.

- Create a queue – createQueue()
- Insert an element – insertElement()
- Remove an element – removeElement()
- Examine an element – peekElement()
Let’s take a look at how these operations might appear in the model given in Figure 3.15. The *insert* operation adds the element to the rear of the queue and moves the rear reference to the next open position. The *get* returns the element at the front of the queue and moves the front reference to the next available element. The *peek* simply returns the value without moving the front reference.

As a first step in designing the data type, we’ll begin with the high level functionality. Once again, we’ll make the simplifying assumption that we are not managing error conditions although we will manage the boundary conditions. Our first task is to create the queue, thus...

**Creating a Queue**

Creating a queue in an embedded context has the same difficulties that were encountered with the linked list. That problem is the general lack of support for dynamic memory allocation. Although supported by the C language, utilizing such capability can have some serious, negative effects on applications with tight timing constraints. As with the linked list, without dynamic memory allocation, the *head* of the queue and the elements of the *body* of the queue must be created at compile time.

**Inserting an Item into a Queue**

Add an element to the rear of the queue. The operation, often called *enqueue*, succeeds unless the queue is full. As input, we have the item to be entered into the queue. Prior to inserting the element, one must ensure that the queue is not full. If the operation succeeds, the queue has a new element at the rear.
Removing an Item from a Queue

Removing and returning the element at the front of queue is often called dequeue. The operation succeeds unless the queue is empty. Prior to removing the element, one must ensure that the queue is not empty. Following the operation, the item at the head of the queue is removed and a copy returned.

Look at an Item at the head of a Queue

The operation succeeds unless the queue is empty. Thus, prior to executing the operation, one must ensure that the queue is not empty. Following the operation, a copy of the first element is returned; the queue remains unchanged.

Observe that there is no direct access to the elements in the queue. One cannot index to a particular data item and there is no way to traverse the collection.

3.4.2 Detailed Design and Building the Queue

The real world examples give some insight into how clients might use the queue data type. From the operations we’ve seen, the public interface naturally follows. As was done with the linked list, we will work with integers, other types follow similarly.

3.4.2.1 Designing the Queue

The public interface to the queue, as illustrated in the class diagram in Figure 3.16, will support the following methods.

- createQueue()
- insertItem()
- getItem()
- peek()

createQueue( ) Method

The head of the queue is created at compile time.

insertItem( ) Method

**Assumptions**
The queue is not full.

**Algorithm**
The pseudo code for the insertItem() method is given in Figure 3.17.
getItem( ) Method

Assumptions
The queue is not empty.

Algorithm
The getItem() method pseudo code is given in Figure 3.18.

```
if the queue is not empty
    remove item from front
    increment front pointer
    return copy of item
else
    return error
end if
```

Figure 3.18.
getItem() Method

peek( ) Method

Assumptions
The queue is not empty.

Algorithm
The pseudo code for the peek() method follows in Figure 3.19.

```
if the queue is not empty
    return copy of item from front of queue
else
    return error
end if
```

Figure 3.19.
peek() Method

3.4.2.2 Implementing the Queue

Implementing the queue is a bit different from the linked list. In contrast to the linked list, the queue simply expresses a collection of access methods to an arbitrary underlying container. Those methods implement the public interface to the container. Two immediate choices for the underlying data structure are the array and the linked list.

We will build the basic queue data type using an array as the internal container or data structure. We will leave the linked list implementation as an exercise. We implement the queue as an interface to the array as illustrated in the accompanying UML diagram in Figure 3.20.

Here we implement the queue as an integer container. Other types fol-
low naturally. Here’s a thought question, ‘How would the example implementation be modified to support enqueuing of any arbitrary data type?’

We specify the size of the queue to be MAXSIZE by specifying the size of the internal array at compile time. The references to the head and tail of the queue are implemented as indices into the array. When an instance of the queue is declared, the head and tail indices will have the same value, thus, indicating that the queue is empty. Each time a value is inserted, the tail index is incremented and each time a value is removed, the head index is incremented.

If one continues to enter items into the queue without removing any, eventually, the tail index is going to be at the end of the array and the array will be full. If we’ve been removing items from the queue, then we’re going to find that both indices are at the top of the array – with the same value thereby stating that the queue is empty yet we will not be able to put anything in. The solution in each case is to start over.

To start over, we simply move the tail (rear) pointer back to the bottom of the array. We do the same thing with the head (front) pointer when necessary. We have now made the queue circular. The queue operates as shown in the following sequence of drawings in Figure 3.21.

In Figure 3.21, we begin with the front and rear (head and tail) pointers at the positions shown. The request to insert 9 is handled by placing the 9 at the position in the array indicated by the rear pointer. The index is incremented mod the array size which effectively moves the pointer or index to the physical start of the queue as shown in the second graphic. The request to insert 3 again places the value at the location identified by the rear pointer and increments that pointer mod the queue size as shown in the third graphic. The front reference will follow similarly as elements are read.

In the design that follows, observe that the circular indexing is implemented by incrementing the index mod the queue size as expressed in the pseudo code fragment.

\[
\text{index} = (\text{index} + 1) \mod \text{QUEUESIZE};
\]

implemented as

\[
\text{index} = (\text{index} + 1) \% \text{QUEUESIZE};
\]

**Caution:** When incrementing the rear pointer, ensure that it does not pass the front pointer.
The code fragment in Figure 3.22 gives an implementation of the queue data type.

```c
#include <stdio.h>

#define MAXSIZE 10

typedef struct {
    int* myDataPtr;
    int head;
    int tail;
    int size;
} Queue;

int insertItem(Queue* aQueuePtr, int aValue);
int getItem(Queue* aQueuePtr);
int peek(Queue* aQueuePtr);

void testQueue(Queue* aQueuePtr);

int main(void)
{
    Queue myQueue = {myData, 0, 0, 0};
    testQueue(&myQueue);
    return;
}

int peek(Queue* aQueuePtr)
{
    int retVal = -1;
    if (aQueuePtr->size > 0)
    {
        // if the queue isn't empty
        retVal = *((aQueuePtr->myDataPtr) + aQueuePtr->head);
    }
    return retVal;
}
```

Figure 3.22
Implementing a Queue Data Type
// implement the insertItem access function
int insertItem(Queue* aQueuePtr, int aValue)
{
    int error = -1;
    // if the queue isn't full
    if (aQueuePtr->size < MAXSIZE)
    {
        // test size before incrementing
        aQueuePtr->size++;
        // put the new item at the tail and wrap if at end of queue
        aQueuePtr->tail = ((aQueuePtr->tail)+1) % MAXSIZE;
        *((aQueuePtr->myDataPtr) + aQueuePtr->tail) = aValue;
        error = 0;
    }
    // print for test
    printf("%d \n", *((aQueuePtr->myDataPtr) + aQueuePtr->tail) );
    return error;
}

// implement the getItem access function
int getItem(Queue* aQueuePtr)
{
    int retVal = -1;
    if (aQueuePtr->size > 0)
    {
        // if the queue isn't empty
        aQueuePtr->size--;
        aQueuePtr->head = ((aQueuePtr->head)+1) % MAXSIZE;
        retVal = *((aQueuePtr->myDataPtr) + aQueuePtr->head);
    }
    return retVal;
}

Figure 3.22 cont.

Implementing a Queue Data Type
3.5 The Stack Data Type

We’ll now examine one final container type that is commonly found in embedded applications, the stack. Like the queue, the stack is an interface expressing stack behavior wrapped around any of a variety of basic containers such as an array or linked list. While the queue permitted one way access at both ends of the data structure, the stack can be accessed only at one end called the top.

Like the queue, the stack is a familiar model for many things that we routinely encounter. Many card games utilize a stack structure of one form or another. The only card visible is the one on the top. The pile of papers on our desk or critical chores that must be done certainly are stack like in nature. The queue implemented a FIFO – First In First Out – protocol; the stack provides LIFO – Last In First Out – behavior.

Let’s look at a couple of simple examples of a stack. Games are always a good place to start. We used this one as children; today it’s a good computer science model for recursion. As illustrated in Figure 3.23, the Towers of Hanoi game provides three stacks. We begin with three pegs. The object of the puzzle is to move the disks from one peg, A, to a second peg, B. The rules are simple. One can move only 1 disk at a time and a large disk can never be placed on top of a smaller one. That is, as one selects a disk to move, access is permitted only to the last disk placed on a peg.

This line of text that I am currently typing represents a stack to the editor that I am using. Without using the mouse, I typically only have access to the last character that I
type. If I make a spelling error or want to change what I’ve typed; pressing the delete key only removes the last character. I get the same behavior if I decide to undo a number of changes. For each undo that I select, I only remove the last change. Last in, first out behavior.

Let’s now begin to formalize the model of the stack. The data type is a homogeneous collection as seen in the other containers; all contained items are of the same type. As with the other containers, we’ll focus our discussion on integers, but the ideas extend naturally.

From an external point of view, the stack has two important elements: the top which is the uppermost element of stack and the first to be removed and the bottom or lowest element of stack. The bottom will be the last element to be removed. The elements are always inserted and removed from the top (LIFO).

### 3.5.1 Abstract Stack Operations

When designing the stack, once again we must think about the functionality or operations that must be supported. These operations are our public interface. They define the behavior of our system. We begin with the use case diagram for the stack data type as given in Figure 3.24.

### 3.5.2 Defining Operations

Once again, from the use case diagram, we identify the minimal set of requirements that the stack should support.

- Create a stack – createStack()
- Insert an element – push()
- Remove an element – pop()
- Examine an element – peek()

Notice that the public interface to the stack looks a bit like a queue. With the queue, we had to manage two references: the front and the rear. With the stack, we are only concerned with the top – which is similar to the rear of the queue. One can express these operations graphically rather easily as we see in Figure 3.25.

Perhaps a few words might be helpful to clarify what is happening here. We begin with a stack with five empty slots as shown in the far left graphic in the figure. The reference, called the stack pointer, refers to the top of the stack. That is the spot where one can make the next entry. When the first element, 3 in this case, is entered, the stack pointer moves to the next free location. This is reflected in the second graphic in the figure. Such an operation is called a push. The element is pushed onto the stack. When we push the next element, the element is inserted and the stack pointer moves once again.

The get operation removes the top element from the stack and returns a copy. The stack pointer is decremented at the same time. Such behavior is evident in the fifth graphic in the figure. The get operation is called a pop. As with the queue, the peek operation simply returns a copy of the element at the top of the stack without modifying the stack pointer.
Observe in the graphic in Figure 3.25 that the get operation returns a copy of the element on the top of the stack and moves the stack pointer. The value, the bit pattern, that was stored there does not disappear; however, it is no longer valid. The next insert (push) will overwrite it.

If the figure were turned upside down, then the stack pointer would be moving the opposite direction. When the stack is built in physical memory, it can be implemented such that the stack pointer moves towards higher memory addresses as data is pushed or so that it moves towards lower memory addresses. It is always important to check the compiler documentation and microprocessor manuals to understand how the stack is implemented on a specific system.

As with the earlier designs, we start with the high level functionality; again, we make the simplifying assumption that we are not managing error conditions although we will manage the boundary conditions. The first task is to create the stack, thus,

Creating a Stack

Creating a stack repeats the earlier challenges with dynamic memory in an embedded context, so the reference to the stack will be created at compile time.

Inserting an Item into a Stack

Adding an element to the top of the stack (often called push) succeeds unless the stack is full. As input, we have the item that is to be entered into the stack. One must ensure that the stack is not full prior to inserting the element otherwise we get a condition called overflow. If the operation succeeds, the stack has a new element at the top and the stack pointer has moved to the next empty spot.

Removing an Item from a Stack

Removing and returning the element at the top of the stack succeeds unless the stack is empty and is often called pop. Prior to removing the element, one must ensure that the stack is not empty. Following the operation, a copy of the item on the top of the stack is returned and the stack pointer is decremented.

Look at an Item at the top of a Stack

The operation succeeds unless the stack is empty. Prior to executing the operation, one must ensure that the stack is not empty. Following the operation, a copy of the first element is returned; the stack pointer remains unchanged.

Observe that there is no direct access to the elements in the stack. One cannot index to a particular data item. One can not iterate over the collection.
3.5.3 Detailed Design and Building the Stack

From the examples and abstract model, we now have some insight into how clients might use the stack data type. As with the queue, the public interface follows naturally from these operations. Once again the implementation will use ints.

3.5.3.1 Designing the Stack

The public interface to the stack will support the methods illustrated in the class diagram in Figure 3.26.

- createStack( )
- push( )
- pop( )
- peek( )

createStack( ) Method

The head of the stack is created at compile time.

push(item) Method

Assumptions
The stack is not full.

Algorithm

The pseudocode for the push() method is shown in Figure 3.27.

```plaintext
if the stack is not full
    insert at the top
    increment stack pointer
else
    return error
end if
```

Figure 3.27 push() Method

pop( ) Method

Assumptions
The stack is not empty.

Algorithm

The pop() method pseudocode is given in Figure 3.28.

```plaintext
if the stack is not empty
    copy of item from top
    decrement stack pointer
    return copy of item
else
    return error
end if
```

Figure 3.28 pop() Method
peek( ) Method

Assumptions
The stack is not empty.

Algorithm
The pseudocode for the peek() method is given in Figure 3.29.

```
if the stack is not empty
    return copy of item from top of stack
else
    return error
end if
```

Figure 3.29
peek() Method

3.5.3.2 Implementing the Stack

As with the queue, there are several possible choices for the internal container (data structure). Alternatives include the array or the linked list. Since we used the array as the underlying container for the queue, let’s build the stack based upon the linked list. As with the queue, the stack functionality is implemented as an interface to a linked list as is shown in the UML diagram in Figure 3.30.

The linked list is managed through a top pointer that points to the first node in the list. New data is entered by adding a new link to the beginning of the linked list. What advantages does this give us? How difficult is such an addition? How does such a scheme influence our ability to create a dynamic stack? Such an implementation helps to ensure that we don’t overflow the stack. Certainly this is not the only implementation. One could have just as easily elected to make the top be the end of the list. The important thing is to document a way and stick with it.

An implementation of the stack data type is presented in the code fragment in Figure 3.31. The design assumes that the dynamic memory allocation functions malloc and free are not available and that a store of free nodes is created at compile time. The free node store is accessed by the functions getNode() and freeNode() which acquire a node when necessary and return it when it is no longer needed.
```c
#include <stdio.h>

// Implement a simple stack data structure using a basic linked list
// Declare the underlying data structures and access function prototypes

// Declare the Node
typedef struct NodeDef
{
    int nodeData;
    struct NodeDef* next;
} Node;

// Declare the Stack
typedef struct
{
    Node* topPtr;
} Stack;

// define the stack access functions
// push an item onto the stack
void push(Stack* aStackPtr, int data)
{
    // get a node from free store
    Node* tempPtr = getNode(data);

    if (aStackPtr->topPtr != NULL)
    {
        tempPtr->next = aStackPtr->topPtr;
    }
    aStackPtr->topPtr = tempPtr;
    return;
}

//  peek at the top of stack
int peek(Stack* aStackPtr)
{
    int tempData = 0;
    if (aStackPtr->topPtr != NULL)
    {
        tempData = (aStackPtr->topPtr)->nodeData;
    }
    return tempData;
}

//  pop an item from the top of stack
int pop(Stack* aStackPtr)
{
    int tempData = 0;
    if (aStackPtr->topPtr != NULL)
    {
        tempData = (aStackPtr->topPtr)->nodeData;
        // return a node to free store
        freeNode(aStackPtr);
    }
    return tempData;
}
```

Figure 3.31 Implementing the Stack Data Type
3.5.4 Common Errors

When working with stacks, one should be aware of two of the more common errors. Both of entail exceeding the bounds of the stack. The first, stack underflow, results from trying to remove an item from an empty stack. The opposite, stack overflow, results from trying to add an item to a full stack.

Always be certain to check the boundary conditions on your containers.

3.6 Algorithms Searching and Sorting

Two of the more important tasks that we’ll encounter when developing certain kinds of applications are searching and sorting. Search is finding something in a set of data while sorting entails putting a set of data in a specified order. Typically these are thought of in the context of large database applications rather than in the traditional embedded world. Increasingly today, embedded applications are starting to utilize databases. Some common applications can include telecommunications equipment, set-top boxes, various consumer products, or medical devices.

Medical devices are certainly one area where searching can be essential. As portable defibrillators move out of the hands of trained medical personnel and into the hands of the general public, the decision of when or if to apply the shock must become more automatic. The ability to match a patient’s ECG against known patterns can be used to ensure that the correct therapy is delivered at the proper time.

An important consideration when we implement any searching or sorting algorithm in an embedded application is the amount of memory necessary to perform the operation. Algorithms that require at least as much memory as the original container find limited utility. The preferred choices are those that can be done in place. That is, in their original container.

3.7 Search

As the problems faced by embedded systems continue to grow in size and complexity, so does the need for relevant information to help solve them. The search problem thus reduces to one of collecting, tagging, and efficiently storing domain specific information in the system memory then subsequently recovering that information as quickly and accurately as possible. Many of these interesting techniques are beyond the scope of this introductory tutorial. The goal in the next few pages is to introduce the problem and some of the terminology. There is wealth of excellent literature on the topic and those wishing to explore in greater depth are encouraged to take advantage of it.

Searching in an embedded context has constraints not generally found with outside world applications. Embedded applications very often demand high performance under tight time and memory constraints. Such constraints make developing and executing good search algorithms much more challenging.

A simple search is binary; the information being sought is either found or it isn’t. A complex search can be fuzzy; more relevant today is the degree of a match. Such searches which can produce results reflecting no match or a range from partial to complete match are becoming more common. When one is considering the degree of match against the search criteria, the search algorithm is most definitely an interesting problem. How would
we implement such a thing? The possibilities for embedded applications in this area are limitless.

We’ll begin with the simple linear search algorithm as a predecessor to the binary search. If the search space is small, linear search offers a reasonable approach. There are, however much more effective techniques.

### 3.7.1 Linear Search

Linear or serial search is the simplest search. It’s the easiest to write and is widely applicable to many problems with smaller data sets.

#### 3.7.1.1 Implementation

The linear search algorithm is given in Figure 3.32.

```c
// Return index of x if found, or -1 if not
int find (int A[], int size, int x)
{
    unsigned char gotIt = -1;
    unsigned int i;
    for ( i = 0; i < size && gotIt < 0; i++ )
    {
        if ( A[i] == x )
            gotIt = i;
    }
    return gotIt;
}
```

For a given array A of N integers, the code fragment in Figure 3.33 executes a linear search for an element x.

#### 3.7.1.2 Analysis

How efficient is the linear search? Let’s look. We’ll specify the search space to be of size N. Clearly the best case occurs if the target is the first element in the array. At the opposite extreme, we have the case in which the element is not in the array. To be able to make such a determination, one must search the complete array. Thus, the worst case per-
formance requires a search of the entire container. The average case finds the element in the middle. Still, that’s a lot of searching.

Linear search is referred to as an exhaustive algorithm; it’s also called the British Museum algorithm.

### 3.7.2 Binary Search

The binary search algorithm, included in ANSI/ISO C as `bsearch()`, is much faster than linear search. Why is this so? The answer is rather simple really. If the array or container is sorted, one can search more quickly. The binary search algorithm takes advantage of this property. Observe that at each step, linear search throws out one element; because the container is sorted, binary search can discard half of the container as we see in the algorithm in Figure 3.34.

#### Algorithm 1 - Binary Search

Start search in middle of array
If x matches the middle element done
else if x is less than middle element search (recursively) in lower half
else x is greater than middle element search (recursively) in upper half

Figure 3.34

The Binary Search Algorithm

#### Example 3.0

To see how the algorithm works, let’s examine the steps shown in Figure 3.35 to search for the number 26 in the sorted set of 15 numbers. We find the middle of the set at position 8. Does it matter if the set contains an even number of elements?

![Binary Search Diagram](image-url)
3.7.2.1 Implementation

A recursive implementation of the binary search algorithm, as shown in the code fragment in Figure 3.36, is rather straight forward.

```
unsigned int binarySearch(int a[], int element, int lower, int upper)
{
    unsigned int gotIt = -1;
    int mid = (lower + upper) / 2;

    if (!((lower> upper))
    {
        if (element == a[mid])
            gotIt = mid;
        else if (element < a[mid])
            gotIt = binarySearch(a, element, lower, mid-1);
        else
            gotIt = binarySearch(a, element, mid+1, upper);
    }
    return gotIt;
}
```

Figure 3.36
A Recursive Implementation of the Binary Search Algorithm

3.7.2.2 Analysis

How well did we do? Let’s take a look. The best case is if the target is found on the first comparison. If that first test fails, then we analyze the opposite boundary. What is the greatest number of recursive calls that can occur? Each call discards at least half of the remaining input and recursion ends when input size is 0. The problem reduces to determining how many times one can divide N in half? The answer is $1 + \log_2 N$. We see this in the following back of the envelop proof. It’s hardly formal, but, sufficient for now.

<table>
<thead>
<tr>
<th>N</th>
<th>Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1 + 1</td>
</tr>
<tr>
<td>4</td>
<td>1 + 2</td>
</tr>
<tr>
<td>8</td>
<td>1 + 3</td>
</tr>
</tbody>
</table>

As noted earlier, an implementation of the binary search is included as a part of the ANSI/ISO C package. The prototype for the function is given in Figure 3.37. We see that bsearch() takes a container (specifically, an array) of a specified size, a function for com-
paring two elements in the container, and the size of an element. To use `bsearch()`, the library `<stdlib.h>` must be included.

```c
void* bsearch(
    void* containerPtr,  // pointer to the container
    const void* element, // pointer to the element
    int contSize,        // size of the container
    int elemSize;        // size of the element
    int (*cmpFctnPtr)(const void* elem0, const void* elem1));  // comparison function
```

There are several interesting things about this prototype.

- To make the algorithm independent of the type of the elements in the container, the type information has been taken away. The pointer to the container must be cast to a void* before `bsearch()` is invoked.
- The same philosophy of generality is carried forward with the requirement that a comparison function be provided. Through such a function, non-integral types, with no inherent notion of equal (such as structs), can be compared. Here, once again, the type information is removed.
- The parameters to the comparison function are qualified as const. Such a qualification prevents the values of the parameters from being changed.

### 3.8 Sorting

Let’s now take a look at the problem of sorting. Sorting involves arranging items into some specified order. Thus, the process should really be called ordering rather than sorting. Sequencing might be another way of describing it. We use sorting as an essential aid to searching. In particular, we’ve seen that binary search requires a sorted input array.

#### 3.8.1 Sorting Algorithms

In the literature, one finds that there are many different sorting algorithms, with many different characteristics,

- Some work better on small input sets others on large input sets – in embedded applications, generally, we are working with smaller input sets although this is changing.
- Some preserve relative ordering of “equal” elements (stable sorts).
- Some need extra memory, some are in place – we prefer in place sorting algorithms for embedded applications.
- Some are designed to exploit data locality (not jump around in memory).

We’ll look at two different sorting algorithms, selection sort and quicksort. Both can be implemented as in place sorts. Quicksort does require some careful pointer/index management to do so, however.
3.8.2 Selection Sort

Selection sort is the sorting equivalent of linear search. It’s simple and is performed exactly as we might do if we have to sort something by hand. The approach is to make repeated passes through the container to be sorted. During each pass, we select the smallest and move that item to the front of the container then repeat the process for the remaining elements. After n-1 passes, where n is the number of elements in the container, everything is sorted. The smallest element is at the top and the largest at the bottom. Clearly the algorithm could sort in the reverse order and conclude with the largest element at the head. The selection sort algorithm is given in Figure 3.38.

Algorithm 2 – Selection Sort

Select the first unsorted element as the smallest in the container
For each unsorted element in container
    Compare against the smallest
    If smaller found
        Replace smallest index with current index
Repeat until no more elements remain

Figure 3.38
The Selection Sort Algorithm

Example 3.1

The diagram in Figure 3.39 shows how selection sort would work on a container of integers.

In the initial container, we select the first element, 6, as the smallest. The comparison process immediately identifies the element 1 as smaller. That becomes the reference element against which others are compared. After all elements to its right have been tested and none found to be smaller, element 1 is interchanged with element 6.

Comparison resumes in the second slot with 6 again. On this pass, 3 is found to be the smallest element and it replaces 6 in the second position. The process repeats for the remainder of the list.

Observe that the smaller elements are perking to the left and the larger to the right.
### 3.8.2.1 Implementation

One can implement selection sort according to the code fragment in Figure 3.40. The function, `findSmallest()` follows similarly.

```c
void selectionSort (int a[], int lower, int upper)
{
    // declare some working variables
    int low = 0;
    int smallest;
    int working;

    for (low = lower; low<upper; low++)
    {
        // find index of smallest element
        smallest = findSmallest(a, low, upper);

        // swap smallest with current top of container
        working = a[low];
        a[low] = a[smallest];
        a[smallest] = working;
    }
    return;
}

int findSmallest(int a[], int lower, int upper)
{
    int smallIndex = lower;
    int i = 0
    for (i=lower+1; i<=upper; i++)
    {
        if a[i] < a[smallIndex])
            smallIndex = i;
    }
    return smallIndex;
}
```

Figure 3.40
Selection Sort Implementation

### 3.8.2.2 Analysis

Let’s make a quick estimate of the performance. First, the main loop in `selectionSort()` iterates \( N \) times. How much work is done each time?

- **By `findSmallest()`**
  
  For each loop iteration, one must search the remaining unsorted numbers. However, the amount of unsorted data is getting smaller.
  
  Worst case, one will have to examine \( n \) elements each time where \( n \) is the number of unsorted elements. Thus, during the first pass \( N-1 \) elements will be examined. For the second, \( N-2 \) will be examined and so on.

- **swap**
  
  For each loop iteration, one must execute one swap. This means that there will be \( N-1 \) swaps.

  \[
  \text{Total Operations} = (N - 1 + 1) + (N - 2 +1) + (N - 3 +1) + \ldots + 1
  \]
The total number of operations, given by the sum of the first N numbers, is,

\[ Total = \frac{N(N + 1)}{2} \]

### 3.8.3 Quicksort

The Quicksort algorithm, discovered/developed by C.A.R. (Anthony) Hoare (1962), is a divide and conquer algorithm. In most cases, for a large container, Quicksort is the fastest known sorting algorithm. The algorithm is included in ANSI/ISO C as qsort().

The basic idea underlying the algorithm is to split the container to be sorted into two parts around a central element or pivot and then place all the elements less than the pivot to its left and all elements that are larger to its right. Next recursively sort those two halves; finally merge the two halves back together again. Let’s begin with a look at the pseudo code for the algorithm in Figure 3.41.

**Algorithm 3 – Quick Sort**

Split (Partition) the container in half

Pick an element \( midval \) of container // The element will become our pivot // The actual value is irrelevant // This step is the key to the process; // it’s not a sorting per se since order is not important

Partition the container into two portions, such that
All elements less than or equal to \( midval \) are left of it, and
All elements those greater than \( midval \) are right of it

Recursively sort each of those 2 portions

Combine the two halves // They’re already in order

Figure 3.41

The Quicksort Algorithm

Let’s save one thousand words and look at a picture…Figure 3.42 in the next example.

Example 3.2

We begin with a container holding the following set of integers.

Before the Partition:

\[ 6 \ 4 \ 2 \ 9 \ 5 \ 8 \ 1 \ 7 \]

The algorithm requires that a pivot element be selected. Ok, since the choice is arbitrary, pick 6.

After the partition:

What values are to the left of the pivot? ⇒ 1 5 2 4 6

What values are to the right of the pivot? ⇒ 6 9 8 7

What about the exact order of the partitioned array? It’s irrelevant. It doesn’t matter? No.

Is the container now sorted? Not yet.
Is it "closer" to being sorted? Yep.
What is the next step? Do it again.
How many more times? Until we're done... until the problem has been decomposed into containers holding a single element. Then put it all together again.

3.8.3.1 Analysis

Best Case for Quicksort

For the best case, we assume *partition()* will split array exactly in half. Under such circumstances, the depth of the recursion is then $\log_2 N$.

Worst Case for Quicksort

If one is very unlucky, one can potentially get the situation in which each pass through partition removes only a single element. In this case, we have $N$ levels of recursion.

Average Case for Quicksort

As a third alternative, one can assume an ‘average’ performance computation on the algorithms. Such an assumption is valid if one considers a large number of sorts and argues that the data is not going to consistently be ordered such that one gets either the best or worst case. Forming the estimate is left as an exercise.

3.8.3.2 Implementation

As noted earlier, an implementation of Quicksort is a part of the ANSI/ISO C package. The prototype for the function is given in Figure 3.43. We see that *qsort()* takes a container (an array) of a specified size, a function for comparing two elements in the
container, and the size of an element. The parameters, specifically the comparison function, mirror the thinking described with respect to the `bsearch()` function. Like the search function, to use `qsort()`, the library `<stdlib.h>` must be included.

```c
void qsort(
    void* containerPtr, // pointer to the container
    int contSize, // size of the container
    int elemSize;        // size of the element
    int (*cmpFctnPtr)
        (const void* elem0, const void* elem1)); // comparison function
```

Figure 3.43
Prototype for the qsort Algorithm

### Example 3.3

The code fragment in Figure 3.44 gives a simple example of using the quicksort algorithm to sort a small array.

```c
#include <stdio.h>
#include <stdlib.h>

int comp0(const void* i0, const void* i1);
int main(void)
{
    int i = 0;
    int myArray[] = {6,4,2,9,5,8,1,7};
    qsort(myArray, 8, sizeof(int), comp0); // sort the container
    for (i = 0; i < 8; i++) // confirm the sort
    {
        printf("%i\n", myArray[i]);
    }
    return 0;
}

int comp0(const void* i0, const void* i1)
{
    int value = 0;
    if (*(int*)i0 > *(int*)i1)
        value = 1;
    else if(*(int*)i0 < *(int*)i1)
        value = -1;
    else
        value = 0;
    return value;
}
```

Figure 3.44
Using Quicksort
3.9 Summary

In this tutorial, we have introduced several basic containers, the linked list, the queue, and the stack. We’ve learned that the linked list supports random access; the queue supports first in first out access; and the stack, last in first out access. We’ve explored how one might proceed to develop the public interface to express the desired behavior of each of the data types then applied the methods to the design and implementation of each. We’ve seen that through such an interface, we can wrap the desired behavior around any other basic containers.

We have also looked at two important processes for working with data. We started with linear and binary search as one approach to the problem of dealing with collections of data. We learned that such searches often assume the data is sorted prior to the search. Such a premise led us to learning to sort the data. Once again, we looked at two alternative methods, selection sort and quicksort and assessed the cost of each.
References


