Lesson 11 : Stacks and Queues

Objectives:

In this chapter you will:

- Learn about stacks
- Examine various stack operations
- Learn how to implement a stack as an array
- Learn how to implement a stack as a linked list
- Discover stack applications
- Learn about queues
- Examine various queue operations
- Learn how to implement a queue as an array
- Learn how to implement a queue as a linked list
- Discover queue applications

Structure of the Lesson:

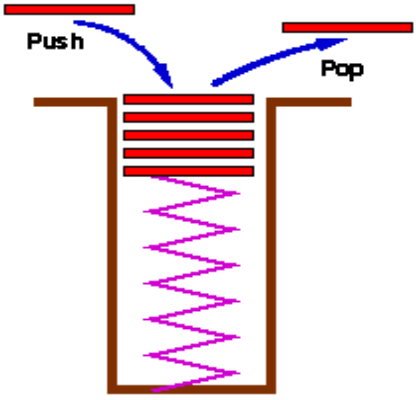
11.1.Introduction

A **data object** is a set of instances or values. The individual instances of a data object are either primitive (or atomic) or composed of instances of another data object. If an individual instance of a data object is not atomic, then its components are called elements. The instances of a data object as well as the elements that constitute individual instances are generally related in some way. In addition to that a set of functions is generally associated with any data object. These functions may transform one instance of an object into another instance of that object, and into an instance of another object also. It may create a new instance without transforming the instances from which the new instance is created.

Boolean, Digit, Letter, NaturalNumber, Integer etc… are examples of data objects. True and False are the instances of Boolean. 0,1,…,9 are the instances of Digit.

A **data structure** is a data object together with the relationships that exist among the instances and among the individual elements that compose an instance.

Stacks and Queues are most frequently used data structures. Both are linear data structures. A stack is called a Last-In First-Out (LIFO) list. Elements are pushed onto the stack at one end, called top of the stack, and removed from the same end. The last element pushed on to the stack is the one to come out first. A common model of a stack is a plate or coin stacker. Plates are "pushed" onto to the top and "popped" off the top.

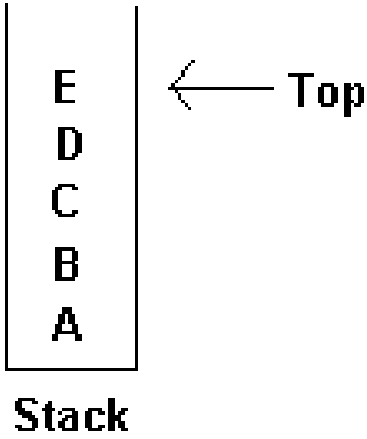


A stack data structure is generally implemented with two principle operations push and pop.

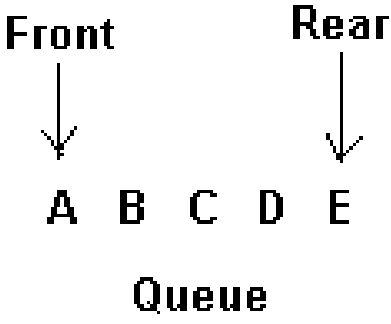
Push : Adds an item to the stack

Pop : Removes the most recently pushed item from the stack.

Given a stack S=(A_1, A_2, A_3, … A_n) then we say that a1 is the bottommost element and element A_i) is on top of element A_{i-1}, 1<i<=n.



The restrictions on a stack imply that if the elements A, B, C, D, E are added to the stack, in that order, then the first element to be removed/deleted must be E. Equivalently we say that the last element to be inserted into the stack will be the first to be removed. For this reason stacks are sometimes referred to as Last In First Out (LIFO) lists.

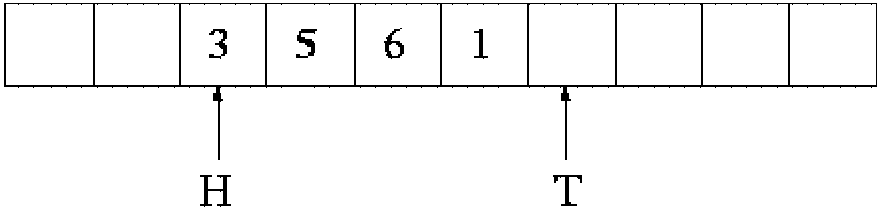


A Queue is First-In First-Out (FIFO) data structure. Elements are inserted into a queue from the 'rear end' and removed from the 'front end'. Given a queue $Q=(A_1, A_2, A_3, \ldots, A_n)$ then we say that A_1 is the first element and the element A_i is in front of the element A_{i-1}, $1<i<=n$.

If the elements A, B, C, D, E are added to the queue, in that order, then the first element to be removed/deleted must be A. Equivalently we say that the first element to be inserted into the queue will be the first to be removed. For this reason queues are referred to as First In First Out (FIFO) lists.

Circular Queue

A circular queue is a particular implementation of a **queue**. It is very efficient and quite useful. A circular queue consists of an array that contains the items in the queue, two array indexes and an optional length. The indexes are called the *head* and *tail* pointers and are labeled H and T on the diagram.



The head pointer points to the first element in the queue, and the tail pointer points just beyond the last element in the queue. If the tail pointer is before the head pointer, the queue wraps around the end of the array.

The problem with circular queue is that, having the head and tail point to the same element would indicate both an empty queue and a full queue. There are two ways around this: either maintain a variable with the number of items in the queue, or create the array with one more element that you will actually need so that the queue is never full.

Insertion and deletion are very simple in a circular queue. To insert, write the element to the tail index and increment the tail, wrapping if necessary (using modulo arithmetic). To delete, save the head element and increment the head, wrapping if necessary (using modulo arithmetic). Instead of using a modulus operator for wrapping (mod in Pascal, % in C) you can use an if statement or (even better) make the size of the array a power of two and simulate the mod with a binary and (& in C).

11.2. Stack Abstract Data Type

As an abstract data type, the stack is a container (data structure) of nodes and has two basic operations: *push* and *pop*. *Push* adds a given node to the top of the stack leaving previous nodes below. *Pop* removes and returns the current top node of the stack. The Stack ADT is given below:

AbstractDataType Stack{
instances
Linear list of elements, one end called top.
operations
Create (): Create an empty stack;
IsEmpty(): Return true if the stack is empty, false otherwise;
IsFull(): Return true if the stack is full, false otherwise;
Top(): Return top element of stack;
Push(x): Place the element x on the top of the stack;

```
        Pop(x) : Remove the top element from stack and
                 assign it to x;
}
```

11.3. Formula based representation of Stack ADT

In formula based representation, a single dimensional array is used to contain the elements of the stack. The following C++ code implements the stack ADT using a linear array.

```
template <class T>
class Stack{
  //LIFO objects.
  public:
      Stack(int maxStacSize=10);
      ~Stack() { delete[] stackList; }
      bool isEmpty() const { return (top == -1); }
      bool isFull() const { return (top == maxSize); }
      T top() const;
      Stack<T>& push(const T& x);
      Stack<T>& pop(T& x);
  private:
      int top, maxSize;
      T  *stackList;
};

template<class T>
Stack<T>::Stack(int maxStackSize)
{// Stack constructor.
      maxSize = maxStackSize – 1;
      stackList = new T[maxStackSize];
      top = -1;
}



template<class T>
```

```
Stack<T>::Top() const
{//Return top element.
        if(isEmpty()) throw OutOfBounds();
        else   return stackList[top];
}

template<class T>
Stack<T>& Stack<T>::push(const T& x)
{// Add x to stack.
        if(isFull()) throw NoMem();
        stackList[++top] = x;
        return *this;
}

template<class T>
Stack<T>& Stack<T>::pop(T& x)
{//pop top element from stack and put it in x.
        if(isEmpty()) throw OutOfBounds();
        x = stackList[top--]; return *this;
 }
```

11.4. Linked representation of Stack ADT

The previous section gave the array representation of stack. Although it is an elegant method, it may lead to wastage of memory space when multiple stacks are to coexist in memory. In such cases stacks can be represented efficiently using a linked list for each stack. The following C++ code gives the linked list implementation of stack.

```
template <class Type>
class Node
{
  public:
        Type info;
```

```
        Node<Type> *link;
};

template<class Type>
class linkedStackType
{
public:
    bool isEmptyStack() const;
      //Function to determine whether the stack is empty.
      //Postcondition: Returns true if the stack is empty,
      //               otherwise returns false.

    void destroyStack();
      //Function to remove all the elements of the stack,
      //leaving the stack in an empty state.
      //Postcondition: stackTop = NULL

    void push(const Type& newItem);
      //Function to add newItem to the stack.
      //Precondition: The stack exists and is not full.
      //Postcondition: The stack is changed and newItem
      //                    is added to the top of the stack.

    Type top() const;
      //Function to return the top element of the stack.
      //Precondition: The stack exists and is not empty.
      //Postcondition: If the stack is empty, the program
      //               terminates; otherwise, the top element
      //               of the stack is returned.

    void pop(Type& item);
      //Function to remove the top element of the stack.
      //Precondition: The stack exists and is not empty.
      //Postcondition: The stack is changed and the top
      //element is removed from the stack.



    linkedStackType();
```

```
    //default constructor
    //Postcondition: stackTop = NULL

private:
   Node<Type> *stackTop; //pointer to the stack

};


template<class Type> //default constructor
linkedStackType<Type>::linkedStackType()
{
      stackTop = NULL;
}

template<class Type>
void linkedStackType<Type>::destroyStack()
{
      Node<Type> *temp; //pointer to delete the node

      while (stackTop != NULL)
            //while there are elements in the stack
      {
        temp = stackTop;
          //set temp to point to the current node
        stackTop = stackTop->link;
         //advance stackTop to the next node
        delete temp;
         //deallocate memory occupied by temp
      }
}// end destroyStack

template<class Type>
bool linkedStackType<Type>::isEmptyStack() const
{
      return(stackTop == NULL);
}
```

```
template<class Type>
void linkedStackType<Type>::push(const Type&
newElement)
{
  Node<Type> *newNode;
       //pointer to create the new node
  newNode = new Node<Type>; //create the node
  assert(newNode != NULL);
  newNode->info = newElement;
      //store newElement in the node
  newNode->link = stackTop;
      //insert newNode before stackTop
  stackTop = newNode;
      //set stackTop to point to the top node
} //end push


template<class Type>
Type linkedStackType<Type>::top() const
{
      assert(stackTop != NULL);
         //if stack is empty terminate the program
      return stackTop->info;     //return the top element
}//end top

template<class Type>
void linkedStackType<Type>::pop(Type& item)
{
  Node<Type> *temp;
    //pointer to deallocate memory
  if (stackTop != NULL)   {
      temp = stackTop;
        //set temp to point to the top node
      stackTop = stackTop->link;
       //advance stackTop to the next node
      item = temp->info;
      delete temp;   //delete the top node
  }
```

```
 else
        cout << "Cannot remove from an empty stack."
             << endl;
}//end pop


    //destructor
template<class Type>
linkedStackType<Type>::~linkedStackType()
{
        destroyStack();
}//end destructor
```

11.5. Stack Applications

Most of the high-level language programs (even C!) make use of a stack frame for the working memory of each procedure or function invocation. When any procedure or function is called, a number of words (together called a *stack frame*) are pushed onto a program stack. When the procedure or function returns, this frame of data is popped off the stack. Like this there are many applications of stack data structure in computer science.

Towers of Hanoi problem – A stack application.

In the Towers of Hanoi problem, you are given n disks and three towers. The disks are initially stacked on tower 1 in increasing order of size from bottom to top. You are to move the disks to tower 2 using tower 3, one disk at a time, such that no disk is ever on top of a smaller one. A solution for this problem using recursive method is given below:

step 1 : move the top n-1 disks from tower 1 to tower 3.

step 2 : move the bottom disk from tower 1 to tower 2.

step 3 : move the n-1 disks from tower 3 to tower 2.

The following is a recursive function to solve Towers of Hanoi problem.

```
void towersOfHanoi(int n, int x, int y, int z)
{
  // Move the top n disks from tower x to tower y.
  // Use tower z fro intermediate storage.
        if(n>0)
        {
          towersOfHanoi(n-1, x, z, y);
          cout<<"move top disk from tower "<<x
               <<" to top of tower "<<y<<endl;
          towersOfHanoi(n-1, z, y, x);
         }
}
```

A non-recursive function can be written for this problem using a formula based stack. The following C++ code gives a non-recursive function to solve the Towers of Hanoi problem.

```
class Hanoi{
   friend void TowersOfHanoi(int);
   public:
        void TowersOfHanoi(int n, int x, int y, int z);
   private:
        Stack<int> *s[4]; // array of pointers to stacks.
};

void Hanoi::TowersOfHanoi(int n, int x, int y, int z)
```

```
{//Move the top n disks from tower x to tower y.
  //Use tower z for intermediate storage.
   int d;  // disk nuber.
   if(n > 0) {
        TowersOfHanoi(n-1, x, z, y);
        s[x]->pop(d);
        s[y]->push(d);
        showstate();
        TowersOfHanoi(n-1, z, y, x);
   }
}


void TowersOfHanoi(int n)
{
   // Preprocessor for Hanoi::towersOfHanoi.
   Hanoi x;
   // create three stacks of size n each.
   x.s[1] = new Stack<int> (n);
   x.s[2] = new Stack<int> (n);
   x.s[3] = new Stack<int> (n);

   for(int d = n; d > 0; d--) // initialize
        x.s[1] ->push(d);  // add disk d to tower 1.
   // move n disks from tower 1 to tower 3
   //using 2 as intermediate.
   x.TowersOfHanoi(n, 1, 2, 3);
}
```

11.6. Queue Abstract Data Type

Queue is a First-In First-Out data structure. Elements are inserted into the queue at the rear end. They are removed from the queue at the front end. The Queue data structure is implemented with two basic operations insert() and delete(). Corresponding to the two **ends** this data structure uses two index variables or pointer variables. The Queue ADT is given below:

AbstractDataType Queue {

instances

ordered list of elements. One end is called *front*, the other *rear*.

operations:

Create(): Create an empty queue;
IsEmpty(): Return true if queue is empty, return false otherwise;
IsFull(): Return true if queue is full, false otherwise;
First() : Return first element of queue;
Last() : Return last element of queue;
Insert(x) : Add element x to the queue;
Delete(x) : Delete front element from queue and put it in x;

}

11.7. Formula Based Representation of Queue ADT

The formula based representation uses an linear array to hold the queue elements, two index variables front and rear and an optional count variable. The following C++ code shows the formula based representation of queue.

```
template<class Type>
class Queue
{
public:

    bool isEmpty () const;
      //Function to determine whether the queue is empty.
      //Postcondition: Returns true if the queue is empty,
      //               otherwise returns false.

    void initializeQueue();
      //Function to initialize the queue to an empty state.
      //Postcondition: count = 0; queueFront = 0;
      //               queueRear = maxQueueSize – 1

   bool isFull () const;
   void destroy ();
      //Function to remove all the elements from the queue.
      //Postcondition: count = 0; queueFront = 0;
      //               queueRear = maxQueueSize – 1

    Type first() const;
      //Function to return the first element of the queue.
      //Precondition: The queue exists and is not empty.
      //Postcondition: If the queue is empty, the program
      //               terminates; otherwise, the first
      //               element of the queue is returned.
    Type last() const;
      //Function to return the last element of the queue.
      //Precondition: The queue exists and is not empty.
      //Postcondition: If the queue is empty, the program
      //               terminates; otherwise, the last
      //               element of the queue is returned.

    void insert(const Type& queueElement);
      //Function to add queueElement to the queue.
      //Precondition: The queue exists and is not full.
      //Postcondition: The queue is changed and
            queueElement is added to the queue.
```

```
    void deleteQ ();
      //Function to remove the first element of the queue.
      //Precondition: The queue exists and is not empty.
      //Postcondition: The queue is changed and the first
      //               element is removed from the queue.

    Queue(int queueSize = 100);
      //constructor
    ~Queue();
      //destructor

private:
    int maxQueueSize; //variable to store the maximum
                      //queue size
    int count;        //variable to store the number of
                      //elements in the queue
    int queueFront;   //variable to point to the first
                      //element of the queue
    int queueRear;    //variable to point to the last
                      //element of the queue
    Type *list;       //pointer to the array that holds
                      //the queue elements
};

template<class Type>
void Queue<Type>::initializeQueue()
{
        queueFront = 0;
    queueRear = maxQueueSize - 1;
        count = 0;
}
template<class Type>
void Queue<Type>::destroy ()
{
        queueFront = 0;
    queueRear = maxQueueSize - 1;
        count = 0;
}
```

```
template<class Type>
bool queueType<Type>::isFull () const
{
  return(count == maxQueueSize);
}

template<class Type>
bool Queue<Type>::isEmpty () const
{
  return(count == 0);
}

template<class Type>
void Queue<Type>::insert(const Type& newElement)
{
  if (!isFull ())   {
       queueRear = (queueRear + 1) % maxQueueSize;
      //use mod operator to advance queueRear because
     //the array is circular
       count++;
       list[queueRear] = newElement;
  }
  else   cout << "Cannot add to a full queue." << endl;
}

template<class Type>
Type Queue<Type>::first() const
{
  assert(!isEmpty());
  return list[queueFront];
}

template<class Type>
Type Queue<Type>::last() const
{
    assert(!isEmpty());
    return list[queueRear];
}
```

```
template<class Type>
void Queue<Type>::deleteQ()
{
  if (!isEmpty())
  {
      count--;
      queueFront = (queueFront + 1) % maxQueueSize;
            //use the mod
            //operator to advance queueFront
            //because the array is circular
  }
  else
      cout << "Cannot remove from an empty queue" <<
endl;
}

//constructor
template<class Type>
Queue<Type>::Queue(int queueSize)
{
   if (queueSize <= 0)    {
      cout << "Size of the array to hold the queue must "
            << "be positive." << endl;
      cout << "Creating an array of size 100." << endl;

      maxQueueSize = 100;
   }
   else
      maxQueueSize = queueSize;
            //set maxQueueSize to queueSize

   queueFront = 0;  //initialize queueFront
   queueRear = maxQueueSize - 1;  //initialize queueRear
   count = 0;
   list = new Type[maxQueueSize];
            //create the array to
            //hold the queue elements
      assert(list != NULL);
}
```

```
template<class Type>
Queue<Type>::~Queue()   //destructor
{
  delete [] list;
}
```

11.8. Linked representation of queue ADT

The linked representation uses a linear linked list to hold the queue elements. Pointers to the first and last nodes in the list are stored in pointer variables front and rear. New nodes are inserted using rear pointer. Nodes are deleted from the queue using front pointer.

```
template <class Type>
class QNode
{
public:
       Type info;
       QNode<Type> *link;
};

template<class Type>
class linkedQueueType
{
public:

    bool isEmpty() const;
      //Function to determine whether the queue is empty.
      //Postcondition: Returns true if the queue is empty,
      //               otherwise returns false.
    bool isFull() const;
      //Function to determine whether the queue is full.
      //Postcondition: Returns true if the queue is full,
      //               otherwise returns false.
```

```
void destroy();
  //Function to delete all the elements from the queue.
  //Postcondition: queueFront = NULL;
  //queueRear = NULL
void initializeQueue();
  //Function to initialize the queue to an empty state.
  //Postcondition: queueFront = NULL;
  //queueRear = NULL

Type first() const;
  //Function to return the first element of the queue.
  //Precondition: The queue exists and is not empty.
  //Postcondition: If the queue is empty, the program
  //               terminates; otherwise, the first
  //               element of the queue is returned.
Type last() const;
  //Function to return the last element of the queue.
  //Precondition: The queue exists and is not empty.
  //Postcondition: If the queue is empty, the program
  //               terminates; otherwise, the last
  //               element of the queue is returned.

void insert(const Type& queueElement);
  //Function to add queueElement to the queue.
  //Precondition: The queue exists and is not full.
  //Postcondition: The queue is changed and
  // queueElement is added to the queue.

void deleteQueue();
  //Function  to remove the first element of the queue.
  //Precondition: The queue exists and is not empty.
  //Postcondition: The queue is changed and the first
  // element is removed from the queue.

linkedQueueType();
  //default constructor
~linkedQueueType(); //destructor
```

```
private:
   QNode<Type> *queueFront; //pointer to the front of
                            //the queue
   QNode<Type> *queueRear;  //pointer to the rear of
                            //the queue
};

template<class Type>
linkedQueueType<Type>::linkedQueueType()      //default
constructor
{
      queueFront = NULL; // set front to null
      queueRear = NULL;  // set rear to null
}

template<class Type>
bool linkedQueueType<Type>::isEmpty() const
{
      return(queueFront == NULL);
}
template<class Type>
bool linkedQueueType<Type>::isFull() const{
      return false;
}
template<class Type>
void linkedQueueType<Type>::destroy()
{
      QNode<Type> *temp;
      while(queueFront!= NULL){
       //while there are elements left in the queue
         temp = queueFront;
             //set temp to point to the current node
         queueFront = queueFront->link;
       //advance first to the next node
         delete temp;
             //deallocate memory occupied by temp
}
   queueRear = NULL;  //set rear to NULL
}
```

```
template<class Type>
void linkedQueueType<Type>::initializeQueue()
{
    destroyQueue();
}

template<class Type>
void       linkedQueueType<Type>::insert(const       Type&
newElement)
{
       QNode<Type> *newNode;

       newNode = new QNode<Type>;  //create the node
       assert(newNode != NULL);

       newNode->info = newElement;    //store the info
       newNode->link = NULL;
           //initialize the link field to NULL
       if(queueFront == NULL)
          //if initially the queue is empty
       {
          queueFront = newNode;
          queueRear = newNode;
       }
       else   //add newNode at the end
       {
          queueRear->link = newNode;
          queueRear = queueRear->link;
       }
}//end insert

template<class Type>
Type linkedQueueType<Type>::first() const
{
       assert(queueFront != NULL);
       return queueFront->info;
}
```

```
template<class Type>
Type linkedQueueType<Type>::last() const
{
	assert(queueRear!= NULL);
	return queueRear->info;
}

template<class Type>
void linkedQueueType<Type>::deleteQueue()
{
  QNode<Type> *temp;

  if(!isEmpty())
  {
	temp = queueFront;
		//make temp point to the first node
	queueFront = queueFront->link;
		 //advance queueFront
	delete temp;
		//delete the first node
	if(queueFront == NULL)
		//if after deletion the queue is empty
	queueRear = NULL;
	     //set queueRear to NULL
  }
  else
	cerr<<"Cannot remove from an empty queue “
	    <<endl;
}//end deleteQueue

template<class Type>
linkedQueueType<Type>::~linkedQueueType()
//destructor
{
	QNode<Type> *temp;
	while(queueFront != NULL)
	 //while there are elements left in the queue
	{
	  temp = queueFront;
```

```
            //set temp to point to the current node
        queueFront = queueFront->link;
            //advance first to the next node
        delete temp;
      //deallocate memory occupied by temp
      }
      queueRear = NULL;  // set rear to null
}
```

11.9. Queue Applications

Railroad Car Rearrangement Problem

Problem description: A freight train has n railroad cars. Each is to be left at a different station. Assume that the n stations are numbered 1 through n and that the freight train visits these stations in the order n through 1. The railroad cars are labeled by their destination. To facilitate removal of the railroad cars from the train, we must reorder the cars so that they are in the order 1 through n from front to back. When the cars are in this order, the last car is detached at each station. The cars are rearranged at a shunting yard that has an input track, an output track, and k holding tracks between the input and output tracks. The following figure shows a shunting yard with k=3 holding tracks H1, H2, and H3. The n cars of the freight train begin in the input track and are to end up in the output track in the order 1 through n from right to left.

To rearrange the cars, we examine the cars on the input track from front to back. If the car being examined is the next one in the output arrangement, we move it directly to the output track. If not, we move it to a holding track and leave it there until it is time to place it in the output. The holding tracks operate in a FIFO manner as cars enter and leave these tracks from the top. When rearranging cars only the following moves are allowed.

1. A car may be moved from the front of the input track into one of the holding tracks or the left of the output track.
2. A car may be moved from the front of a holding track to the left end of the output track.

Solution

When a car is to be moved to a holding track, we can use the following track selection to decide which holding track to move it to. " Move car '**C'** to a holding track that contains only cars with a smaller label; if there are several such tracks, select one with largest label at its left end; otherwise, select an empty track (if one remains).

Program to rearrange cars using queues.

```
void Output (int& minH, int& minQ,
                LinkedQueue<int> H[], int k, int n)
{
  // Move from hold to output and update minH and minQ.
  int c; // car index.
  // delete smallest car minH from queue minQ.
  H[minQ].deleteQ(c);
  cout<< "Move car "<<minH<< " from holding track"
       << minQ << " to output "<< endl;
// find new minH and minQ by checking front of all queues.
  MinH = n + 2;
for( int i=1; i<= k; i++)
  if( !H[I].IsEmpty() && (c = H[I].First()) < minH)
  {
      minH = c;
      minQ = i;
  }
}

bool Hold( int c, int& minH, int &minQ,
            LinkedQueue<int> H[], int k)
{// Add car C to a holding track.
 // Return false if no feasible holding track.
 // Return true otherwise.
```

```
  // find best holding track for car c initialize.

   int BestTrack = 0, BestLast = 0, x;

// scan holding tracks.
for( int i=1; i <= k; i++)
   if(!H[i].IsEmpty())
   {
        x = H[i].Last();
        if( c > x && x > BestLast)
        {
               BestLast = x;
               BestTrack = i;
        }
   }
   else if (!BestTrack) BestTrack = i;
   if (!BestTrack) return false;
        H[BestTrack].insert(c);
   Cout<<" Move car "<< c << " from input "
         << " to holding track "<< BestTrack << endl;
   if(c< minH)
   {
     minH = c;
     minQ = BestTrack;
   }
   return true;
}
```

11.10. Summary

A stack is simply another collection of data items and thus it would be possible to use exactly the same specification as the one used for our general collection. However, collections with the LIFO semantics of stacks are so important in computer science that it is appropriate to set up a limited specification appropriate to stacks only.

Although a linked list implementation of a stack is possible (adding and deleting from the head of a linked list produces exactly the LIFO semantics of a stack), the most common applications for stacks have a space restraint so that using an array implementation is a natural and efficient one.

Like stacks, queues can be used to remember the search space that needs to be explored at one point of time in traversing algorithms. *Breadth-First* search of a graph uses a queue to remember the nodes yet to be visited.

A queue is natural data structure for a system to serve the incoming requests. Most of the process scheduling or disk scheduling algorithms in operating systems use queues. Computer hardware like a processor or a network card also maintain buffers in the form of queues for incoming resource requests. A stack like data structure causes starvation of the first requests, and is not applicable in such cases. A mailbox or port to save messages to communicate between two users or processes in a system is essentially a queue like structure.

11.11. Technical Terms:

Pop
This operation removes the top element of the stack and stores the top element into a location called poppedElement

Push
This operation places a new element on top of the stack

Program stack
A data structure, which the computer uses to implement function calls among numerous other applications

Queue
A data structure in which the elements are added at one end, called the rear, and deleted from the other end

Stack
A data structure in which the elements are added and removed from one end only

Circular Queue
An implementation of queue data structure, in which the first and last positions, in the container array are treated as adjacent.

11.12. Model Questions:

1. Write a C++ program to test the array implementation of stack.
2. Write about different types of queues.
3. What is a double-ended queue?
4. Explain how stack is useful in solving towers of Hanoi?
5. Write a C++ program for linked representation of queue.

11.13. References:

Data Structures, Algorithms, and Applications in C++ by SAHNI.

AUTHOR:

Y. VENKATESWARA RAO, M.C.A.,
Lecturer,
Dept.Of Computer Science,
JKC College,
GUNTUR