|  |  |  |  |
| --- | --- | --- | --- |
| ADT Bag METHODS | Array | Resizable Array | Linked List |
| Int getCurrentSize () | Fixed size | Return numOfE | Return numOfE |
| Boolean isFull() | True-no add | True-MakeRoom() | Can’t be full |
| Boolean isEmpty() | Check a[0]] | Check a[o] | Look at firstNode |
| Void clear() | Clear | Clear | firstNode=null |
| Boolean remove(T an Entry) | End->pos. | End ->pos | Skip link |
| Boolean add (T anEntry) | To end | To end | To beginning |
| T remove() | From end | From end | From beginning |
| Boolean contains(T an Entry) | Search | Search | Search |

Public static final= default constant

ADT LIST= fist position is 1 **ADT BAG IMPLEMENTATIONS**

ADT- Language Independent

**Algorithm Effeciency**  No order, no number of items, no organization, duplicates allowed

N^2+40n-100 = O(n^2)

|  |  |  |
| --- | --- | --- |
| ADT LIST METHODS | Array | Linked List |
| Add(T newEntry) | Adds to end | Adds to end |
| Boolean add(int New Position, T newEntry) | Adds to location N | Adds to location N |
| T remove (int Pos.) | Removes from pos | Removes from pos |
| Clear | numOfEntries=0 |  |
| Boolean replace(int pos, T newEntry) | Pu |  |
| Public T getEntry (int pos) |  |  |
| Boolean contains (T aEntry) |  |  |
| Int getLength() | Return |  |
| Boolean isEmpty() |  |  |

N + log2N +20= O(n)

Nlog2n+ n^2= O(n^2)

Nlog2n + n = O(nlog2n)

**Binary Search**

While(true){

Int mid=(end+start)/2;

If(a[mid==anEntry)

Return true;

Else if (end==start)

Return false

Else if (a[mid]<an Entry)

Start= mid+1

Else

End= mid-1

If(end<0) end=0

**Terms:**

**Vector** – synchronized but ArrayList is not

**TakeHOMEQUIZZES**

In an array implementation of the ADT List, if user remove the item at position 1, it is ok to copy the last item in the list to position 1 instaed of shifting all element to lower indexes. –**FALSE**

MakeRoom()- shifts all elements to higher index

f(n) is O(g(n)= g(n) is **an upperbound** of f(n)

**DownSides to Recursion**

**1)**uses more memory 2) stores data then reloads

3)Base Case +Recursive Case

**Activation Frame**- chunk of memory for local variables

**Runtime Stack**- stack of Activation Frames

Interface is used as superclass

**Implementing a DoublyLInkedList**

if(firstNode == null){

firstNode = new Node(null, newEntry, null); lastNode = firstNode;}

else { {

Node newNode = new Node(lastNode, newEntry, null);

lastNode.next = newNode;

lastNode = newNode; }

**formula to calculate the sum n(n+1)/2**

**forLoop= O(n) | nested forLoop = 0(n^2)**

**binary search=log2n | sequential = n**

**ADT STACK- LIFO**

**ADT BAG**

No rules about the order of stuff you put in

No rules about the number of items you can put in

No rules about how stuff are organized in the bag

No rules about duplicate items

No rules about the type of stuff you can put in

**ADT LIST**

A list is a way to organize a collection of data (same data

type)

A list is never full

Each item in the list has a unique position (similar to array)

The first position is 1 not 0

We can add a new entry at the end of the list.

We can add a new entry anywhere: at the beginning, the

end, or in between items.

We can remove an entry.

We can remove all entries.

We can replace an entry.

We can look at any entry.

We can find out whether the list contains a particular entry.

We can count the number of entries in the list.

We can see whether the list is empty.**REMOVE LINKED LIST ADT LIST**

public T remove(int givenPosition)

T result = null;

if((givenPosition >= 1) && (givenPosition <= numberOfEntries))

assert !isEmpty();

if(givenPosition == 1)

result = firstNode.data;

firstNode = firstNode.next;

if(numberOfEntries == 1)

lastNode = null;

else

Node nodeBefore = getNodeAt(givenPosition - 1);

Node nodeToRemove = nodeBefore.next;

Node nodeAfter = nodeToRemove.next;

nodeBefore.next = nodeAfter;

result = nodeToRemove.data;

if(givenPosition == numberOfEntries)

lastNode = nodeBefore;

numberOfEntries--;

return result;

**POSTFIX EXPRESSION**

(a + b) \* c - (d + e \* f)?

(((a + b) \* c) - (d + (e \* f)))

(((a b +) c \*) (d (e f \*) +) -)

a b + c \* d e f \* + -

REMOVE T(AN ENTRY)

public boolean remove(T anEntry){

int position = getPositionOf(anEntry);

if(position != -1){

list.remove(position);

return true; }

return false; } **Next Method**

private int getPositionOf(T anEntry)

int result = -1;

for(int i = 1; i <= list.getLength(); i++){

if(list.getEntry(i).equals(anEntry)){

result = i;

break; }

}return result;

**Algorithm Analysis**

**Method foo(n)= O(n)**

for(int i = 0; i < n; i++) {

result = result + foo(n);}

**Overall O(n^2)**

20 + 21 + 22 + ... + 2n? = 2^(n+1)-1

**Infix to Postfix**

**a / b ∗ (c + (d − e))**

|  |  |  |
| --- | --- | --- |
| **Next Characters from infix expression** | **Postfix Form** | **Operator Stack**  **Bottom – Top** |
| **A** | **A** |  |
| **/** | **A** | **/** |
| B | **Ab** | **/** |
| \* | **Ab/** |  |
|  | **Ab/** | **\*** |
| ( | **Ab/** | **\*(** |
| C | **Ab/c** | **\*(** |
| + | **Ab/c** | **\*(+** |
| ( | **Ab/c** | **\*(+(** |
| D | **Ab/cd** | **\*(+(** |
| - | **Ab/cd** | **\*(+(-** |
| e | **Ab/cde** | **\*(+(-** |
| ) | **Ab/cde-** | **\*(+** |
| ) | **Ab/cde-+** | **\*(** |
|  | **Abc/cde-\*** |  |

**Scaling**

|  |  |  |
| --- | --- | --- |
| **METHODS** | **ARRAYBAG** | **LINKEDBAG** |
| **Add(newEntry)** | **O(1)** | **O(N)** |
| **Add(newEntry,pos)** | **O(N)** | **O(N)** |
| **Remove(pos)** | **O(N)** | **O(N)** |
| **getEntry(pos)** | **O(1)** | **O(n)** |
| **Replace (pos,ent)** | **O(1)** | **O(N)** |
| **Contains(entry)** | **O(N)** | **O(N)** |