Chapter 6: Stacks

Saturday, February 13, 2021 12:51 PM

6.1 The Abstract Data Type Stack

1. **DEVELOPING AN ADT DURING THE DESIGN OF A SOLUTION**
   - One instance in which stacks are used is when you used the Backspace key to correct your mistakes.
   - ADT operations required for this purpose:
     i. Add a new item to the ADT
     ii. Remove from the ADT the item that was added most recently
     iii. See whether the ADT is empty
     iv. Get the item that was added to the ADT most recently
   - These operations defined he required ADT, which happens to be well-known: It is usually called a stack.

2. **SPECIFICATIONS FOR THE ADT STACK**
   - Identified ADT operations above ^^
   - A stack has the LIFO property (last in, first out)
     i. The queue ADT is the one that as the FIFO property (first in, first out)
   - Refining the specification of the ADT stack:

ABSTRACT DATA TYPE: STACK

DATA

- A finite number of objects, not necessarily distinct, having the same data type and ordered by when they were added.

OPERATIONS

PSEUDOCODE	DESCRIPTION
`isEmpty()`	Task: Sees whether this stack is empty. Input: None. Output: True if the stack is empty; otherwise false.
`push(newEntry)`	Task: Adds `newEntry` to the top of this stack. Input: `newEntry`. Output: True if the operation is successful; otherwise false.
`pop()`	Task: Removes the top of this stack. That is, it removes the item that was added most recently. Input: None. Output: True if the operation is successful; otherwise false.
`peek()`	Task: Returns the top of this stack. That is, it gets the item that was added most recently. The operation does not change the stack. Input: None. Output: The top of the stack.

   - Axioms: a set of mathematical rules that precisely specify the behavior of each operation of an ADT

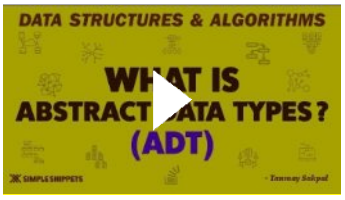
Note: Axioms for the ADT stack

```
(new Stack()).isEmpty() = true
(new Stack()).pop() = false
(new Stack()).peek() = error
(aStack.push(item)).isEmpty() = false
(aStack.push(item)).peek() = item
(aStack.push(item)).pop() = true
```

Note: Axioms for the ADT stack

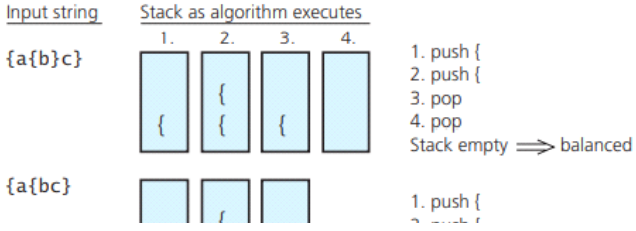
```
(new Stack()).isEmpty() = true
(new Stack()).pop() = false
(new Stack()).peek() = error
```

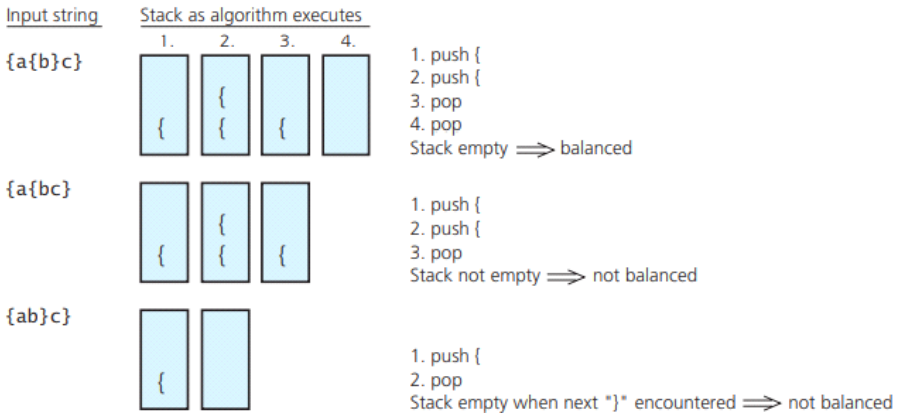
To clarify difference between abstraction and implementation: What is Abstract Data Types(ADT) in Data Structures ? | with Example



6.2 Simple Uses of a Stack

1. **CHECKING FOR BALANCED BRACES**
   a. Traces of the algorithm that checks for balanced braces:





2. RECOGNIZING STRINGS IN A LANGUAGE

$L = \{s\$s' : s \text{ is a possibly empty string of characters other than } \$, s' = \text{reverse}(s)\}$

- Stack is useful for determining whether a given string is in L: suppose you traverse the first half of the string and push each character onto a stack. Then, when you reach the $, you can undo the process: For each character in the second half of the string, you pop a character off the stack. However, you must match the popped character with the current character in the string to ensure that the second half of the string is the reverse of the first half.
- The stack must be empty *only when* you reach the end of the string, otherwise one half of the string is longer than the other so the string is not in L

```
aStack = a new empty stack

// Push the characters that are before the $ (that is, the characters in s) onto the stack
i = 0
ch = character at position i in aString
while (ch is not a '$')
{
   aStack.push(ch)
   i++
   ch = character at position i in aString
}

// Skip the $
i++

// Match the reverse of s
inLanguage = true                  // Assume string is in language
while (inLanguage and i < length of aString)
{
   if (!aStack.isEmpty())
   {
      stackTop = aStack.peek()
      aStack.pop()
      ch = character at position i in aString
      if (stackTop equals ch)
         i++                       // Characters match
      else
         inLanguage = false        // Characters do not match (top of stack is not ch)
   }
   else
         inLanguage = false        // Stack is empty (first half of string is shorter
                                   // than second half)
}
if (inLanguage and aStack.isEmpty())
   aString is in language
else
   aString is not in language
```

- NOTE: you can use an ADT's operations in an application without the distraction of implementation details

6.3 Using Stacks With Algebraic Expressions

1. EVALUATING POSTFIX EXPRESSIONS

a. Your use of an ADT's operations should note depend on its implementation
b. An operator in a postfix expression applies to the two operands that immediately precede it, so the calculator must be able to retrieve the operands entered most recently.
c. Organizing operands and operators:
   i. Each time you enter an operand, the calculator pushes it onto a stack
   ii. Each time you enter an operator, the calculator applies it to the top two operands on the stack, pops the operands from the stack, and pushes the result onto the stack

Key entered	Calculator action		Stack (bottom to top):
2	push 2		2
3	push 3		2 3
4	push 4		2 3 4
+	operand2 = peek	(4)	2 3 4
	pop		2 3
	operand1 = peek	(3)	2 3
	pop		2
	result = operand1 + operand2	(7)	
	push result		2 7
*	operand2 = peek	(7)	2 7
	pop		2
	operand1 = peek	(2)	2
	pop		
	result = operand1 * operand2	(14)	
	push result		14

d. Upon termination of the algorithm, the value of the expression will be on the top of the stack.

2. CONVERTING INFIX EXPRESSIONS TO EQUIVALENT POSTFIX EXPRESSIONS

a. Three important facts when manually converting infix expressions to postfix form:
   i. Operands always stay in the same order with respect to each other
   ii. An operator will move only to the right with respect to the operands; that is, if in the infix expression the operand *x* precedes the operator *op*
   iii. All parentheses are removed

b. NOTE: We should not discard the parentheses of an infix expression because they define an isolated subexpression independently of the rest of the expression and therefore must be evaluated separately

c. Five step process to convert from infix to postfix form:
   1. When you encounter an operand, append it to the output string *postfixExp*
   2. Push each "(" onto the stack
   3. When you encounter an operator, if stack is empty push the operator onto the stack. Otherwise, pop operators of greater or equal precedence from the stack and append them to *postfixExp*. You stop when you encounter either a "(" or an operator of lower precedence or when the stack becomes empty.
   4. When you encounter a ")", pop operators off the stack and append them to the end of postFixExp until you encounter the matching "("
   5. When you reach the end of the string, you append the remaining contents of he stack to postfixExp.

d. Algorithm trace:

ch	aStack (bottom to top)	postfixExp	
a		a	
–	–	a	
(	–(	a	
b	–(	ab	
+	–(+	ab	
c	–(+	abc	
*	–(+*	abc	
d	–(+*	abcd	
)	–(+	abcd*	Move operators from stack to `postfixExp` until "("
	–(	abcd*+	
	–	abcd*+	
/	–/	abcd*+	Copy operators from stack to `postfixExp`
e	–/	abcd*+e	
		abcd*+e/–	

6.4 Using a Stack to Search a Flight Map

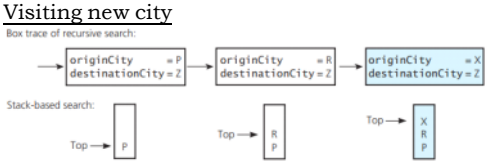
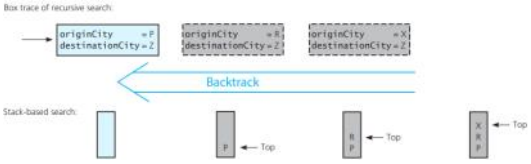
1. Use a stack to organize an exhaustive search
2. Possible outcomes of the exhaustive search strategy:
   1. You eventually reach the destination city.
   2. You reach a city *C* from which there are no departing flights.
   3. You go around in circles forever.
3. Use backtracking to recover from a wrong choice. In order to backtrack, it must maintain information about the order in which it visits the cities. Therefore, it is recommended that you maintain the sequence of visited cities in a stack. --> each time you visit a city, push its name into the stack. If you need to backtrack, just pop a city name from the stack.

```
aStack = a new empty stack
aStack.push(originCity) // Push origin city onto aStack

while (a sequence of flights from the origin to the destination has not been found)
{
   if (you need to backtrack from the city on the top of the stack)
      aStack.pop()
   else
   {
      Select a destination city C for a flight from the city on the top of the stack
      aStack.push(C)
   }
}
```

The stack contains a directed path from the origin city at the bottom of the stack to the city at the top of the stack.

4. for more information, just check the textbook

6.5 The Relationship Between Stacks and Recursion

3 KEY ASPECTS	STACK-BASED SEARCH	RECURSIVE-BASED SEARCH
Visiting new city 	The algorithm searchS visits city C by pushing C onto a stack.	The recursive algorithm searchR visits a new city C by calling searchR(C, destinationCity)
Backtracking 	The algorithm searchS backtracks by explicitly popping from its stack	The algorithm searchR backtracks by returning from the current recursive call.
Termination *(All possibilities are exhausted when, after backtracking to the origin city, no unvisited adjacent cities remain)*	For searchS, no unvisited cities are adjacent to the origin when the stack becomes empty.	This situation occurs for searchR when all boxes have been crossed off in the box trace and a return occurs to the point of the original call to the algorithm.

- Typically, stacks are used to implement recursive functions
- Each recursive call generates an activation record that is pushed onto a stack
- You can use stacks when implementing a non-recursive version of a recursive algorithm

SUMMARY

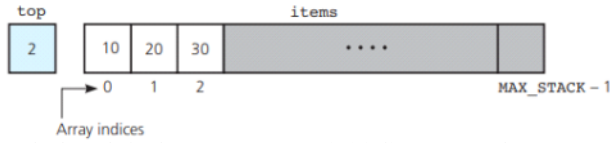
1. The ADT stack operations have a last-in, first-out (LIFO) behavior.
2. Algorithms that operate on algebraic expressions are an important application of stacks. The LIFO nature of stacks is exactly what the algorithm that evaluates postfix expressions needs to organize the operands. Similarly, the algorithm that transforms infix expressions to postfix form uses a stack to organize the operators in accordance with precedence rules and left-to-right association.
3. You can use a stack to determine whether a sequence of flights exists between two cities. The stack keeps track of the sequence of visited cities and enables the search algorithm to backtrack easily. However, displaying the sequence of cities in their normal order from origin to destination is awkward, because the origin city is at the bottom of the stack and the destination is at the top.
4. A strong relationship between recursion and stacks exists. Most implementations of recursion maintain a stack of activation records in a manner that resembles the box trace.
5. The formal mathematical study of ADTs uses systems of axioms to specify the behavior of ADT operations.

Chapter 7: Implementations of the ADT Stack

Thursday, February 18, 2021 10:48 AM

7.1 An Array-Based Implementation

1. Since entries are added or removed at the top of the stack, we can avoid shifting the current entries if we anchor the bottom of the stack at index 0 and track the location of the stack's top using an index *top*. Then if *items* is the array and *items[top]* is the top entry, *items* and *top* can be the private data members of our class of stacks.



2. Include a default constructor to initialize *item* and *top*
3. *NOTE:* because we plan to store a stack's entries in statically allocated memory, the compiler-generated destructor and copy constructor will be sufficient. If we were to use a dynamically allocated array, we would have to define a default and a copy constructor.
4. Header file for array-based stack:

```
const int MAX_STACK = maximum-size-of-stack;

template<class ItemType>
class ArrayStack : public StackInterface<ItemType>
{
private:
   ItemType items[MAX_STACK];     // Array of stack items
   int      top;                  // Index to top of stack

public:
   ArrayStack();                  // Default constructor
   bool isEmpty() const;
   bool push(const ItemType& newEntry);
   bool pop();
   ItemType peek() const;
}; // end ArrayStack
```

5. Implementation file for array-based stack:

```
template<class ItemType>
ArrayStack<ItemType>::ArrayStack() : top(-1)
{
}  // end default constructor

// Copy constructor and destructor are supplied by the compiler

template<class ItemType>
bool ArrayStack<ItemType>::isEmpty() const
{
   return top < 0;
}  // end isEmpty

template<class ItemType>
bool ArrayStack<ItemType>::push(const ItemType& newEntry)
{
   bool result = false;
   if (top < MAX_STACK - 1)  // Does stack have room for newEntry?
   {
      top++;
      items[top] = newEntry;
      result = true;
   }  // end if

   return result;
}  // end push

template<class ItemType>
bool ArrayStack<ItemType>::pop()
```

? What is **statically allocated memory** and **dynamically allocated memory?**
https://stackoverflow.com/questions/8385322/difference-between-static-memory-allocation-and-dynamic-memory-allocation

Static allocation: the memory for your variables is allocated when the program starts. It applies to global variables, file scope variables, and variables qualified with *static* defined inside functions.

Dynamic memory allocation: you can control the exact size and the lifetime of these memory locations.

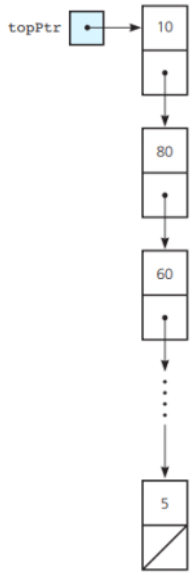
? What is **assert()** in C++?
http://www.cplusplus.com/reference/cassert/assert/

assert(): Expression to be evaluated. If this expression evaluates to 0, this causes an assertion failure that terminates the program.

```
{
   bool result = false;
   if (!isEmpty())
   {
      top--;
      result = true;
   } // end if

   return result;
} // end pop

template<class ItemType>
ItemType ArrayStack<ItemType>::peek() const
{
   assert (!isEmpty()); // Enforce precondition

   // Stack is not empty; return top
   return items[top];
} // end peek
// end of implementation file
```

7.2 A Link-Based Implementation

1. Begin with header file based on *StackInterface* (just like we did for array-based)
2. *topPtr* is a pointer to the head of the linked nodes
3. *NOTE:* Because memory is allocated dynamically for the nodes, you must write both a copy constructor and a virtual destructor for the link-based stack implementation
4. If a shallow copy of the stack is sufficient, you can omit the copy constructor, in which case the compiler generates a copy constructor that performs a shallow copy.
5. We have to write our own copy constructor in order to make a deep copy of the link based stack - otherwise, it would just make a copy *of topPtr* which would point to the same head node as the original stack and the stack itself would not be copied.



6. Header file for link-based stack:

```
template<class ItemType>
class LinkedStack : public StackInterface<ItemType>
{
private:
   Node<ItemType>* topPtr;    // Pointer to first node in the chain;
                              // this node contains the stack's top

public:
//  Constructors and destructor:
    LinkedStack();             // Default constructor
    LinkedStack(const LinkedStack<ItemType>& aStack);  // Copy constructor
    virtual ~LinkedStack();    // Destructor

//  Stack operations:
    bool isEmpty() const;
    bool push(const ItemType& newItem);
    bool pop();
    ItemType peek() const;
}; // end LinkedStack

#include "LinkedStack.cpp"
#endif
```

7. Implementation file for link-based stack: (check textbook - too long to include here)
8. Comparing implementations:
   a. The array-based implementation is a reasonable choice if the number of items in the stack does not exceed the fixed size of the array
   b. For stacks that might be large, but often are not, the array-based implementation will waste storage. In that case the link-based implementation is a better choice.

9. Initializers vs. set methods for constructors:
   a. Use an initializer if the value of the data member has no restrictions
   b. Use a set method if the value of the data member needs validation

7.3 Implementations That Use Exceptions

1. What happens when you try to look at the top of an empty stack? Our implementations deal with the misuse of this method by beginning its definition with the statement: *assert(!isEmpty());*
2. If the stack is empty, *assert* will issue an error message and halt execution
3. Another option is throwing an exception if *peek*'s precondition is violated - if it is called when the stack is empty. Define class *PrecondViolatedExcep* that *peek* can use.
4. Header file for *PrecondViolatedExcep*:

```
#include <stdexcept>
#include <string>

using namespace std;

class PrecondViolatedExcep: public logic_error
{
public:
   PrecondViolatedExcep(const string& message = "");
};  // end PrecondViolatedExcep

#endif
```

5. Implementation file for *PrecondViolatedExcep*:

```
PrecondViolatedExcep::PrecondViolatedExcep(const string& message):
      logic_error("Precondition Violated Exception: " + message)
{
}  // end constructor
```

6. Revise the declarations of peek in both of the header files *ArrayStack.h* and *LinkedStack.h* by adding a *throw* clause as follows:

```
ItemType peek() const throw(PrecondViolatedExcep);
```

7. Similarly in implementation, replace the *assert* statement with this:

```
if (isEmpty())
   throw PrecondViolatedExcep("peek() called with empty stack");
```

8. Final definition of *peek* in *LinkedStack*:

```
template<class ItemType>
ItemType LinkedStack<ItemType>::peek() const throw(PrecondViolatedExcep)
{
   // Enforce precondition
   if (isEmpty())
      throw PrecondViolatedExcep("peek() called with empty stack");

   // Stack is not empty; return top
   return topPtr->getItem();
}  // end getTop
```

SUMMARY

1. You can implement a stack by using an array. If the bottom of the stack is in the first element of the array, no stack entries are moved when you add or remove entries.
2. You can implement a stack by using a chain of linked nodes that has a head pointer. The first node of the chain should contain the top of the stack to provide the easiest and fastest addition and removal operations.
3. You should call `assert` or throw an exception to enforce the precondition for the method `peek`.

Chapter 8: Lists

Thursday, February 18, 2021 3:47 PM

8.1 Specifying the ADT List

1. The list has one first item and one last item. Except for the first and last items, each item has a unique predecessor and a unique successor. The first item (head or front) of the list does not have a predecessor and the last item (tail or end) does not have a successor.
2. Lists contain items of the same type (list of passwords, list of grocery items, list of phone numbers, etc.)
3. *NOTE:* The ADT list is simply a container of items whose order you indicate and whose position you reference by number.
4. ADT List operations:

Pseudocode	Description
`isEmpty()`	Task: Sees whether this list is empty. Input: None. Output: True if the list is empty; otherwise false.
`getLength()`	Task: Gets the current number of entries in this list. Input: None. Output: The integer number of entries currently in the list.
`insert(newPosition, newEntry)`	Task: Inserts an entry into this list at a given position. An insertion before existing entries causes the renumbering of entries that follow the new one. Input: `newPosition` is an integer indicating the position of the insertion, and `newEntry` is the new entry. Output: True if 1 ≤ `newPosition` ≤ `getLength()` + 1 and the insertion is successful; otherwise false.
`remove(position)`	Task: Removes the entry at a given position from this list. A removal before the last entry causes the renumbering of entries that follow the deleted one. Input: `position` is the position of the entry to remove. Output: True if 1 ≤ `newPosition` ≤ `getLength()` and the removal is successful; otherwise false.
`clear()`	Task: Removes all entries from this list. Input: None. Output: None. The list is empty.
`getEntry(position)`	Task: Gets the entry at the given position in this list. Input: `position` is the position of the entry to get; 1 <= `position` <= `getLength()`. Output: The desired entry.
`setEntry(position, newEntry)`	Task: Replaces the entry at the given position in this list. Input: `position` is the position of the entry to replace; 1 <= `position` <= `getLength()`. `newEntry` is the replacement entry. Output: None. The indicated entry is replaced.

5. More info on axioms in the textbook

8.2 Using the List Operations

1. Displaying the items on a list

```
// Displays the items on the list aList.
displayList(aList)

   for (position = 1 through aList.getLength())
   {
      dataItem = aList.getEntry(position)
      Display dataItem
   }
```

2. Replacing an item

```
// Replaces the ith entry in the list aList with newEntry.
// Returns true if the replacement was successful; otherwise return false.
replace(aList, i, newEntry)

   success = aList.remove(i)
   if (success)
      success = aList.insert(i, newItem)

   return success
```

3. Creating a list of names in alphabetical order

```
alphaList = a new empty list
alphaList.insert(1, "Amy")      // Amy
alphaList.insert(2, "Ellen")    // Amy Ellen
alphaList.insert(2, "Bob")      // Amy Bob Ellen
alphaList.insert(3, "Drew")     // Amy Bob Drew Ellen
alphaList.insert(1, "Aaron")    // Aaron Amy Bob Drew Ellen
alphaList.insert(4, "Carol")    // Aaron Amy Bob Carol Drew Ellen
```

8.3 An Interface Template for the ADT List

```
#ifndef _LIST_INTERFACE
#define _LIST_INTERFACE

template<class ItemType>
class ListInterface
```

```
{
public:
   /** Sees whether this list is empty.
    @return  True if the list is empty; otherwise returns false. */
   virtual bool isEmpty() const = 0;

   /** Gets the current number of entries in this list.
    @return  The integer number of entries currently in the list. */
   virtual int getLength() const = 0;

   /** Inserts an entry into this list at a given position.
    @pre  None.
    @post  If 1 <= position <= getLength() + 1 and the insertion is
       successful, newEntry is at the given position in the list,
       other entries are renumbered accordingly, and the returned
       value is true.
    @param newPosition  The list position at which to insert newEntry.
    @param newEntry  The entry to insert into the list.
    @return  True if insertion is successful, or false if not. */
   virtual bool insert(int newPosition, const ItemType& newEntry) = 0;

   /** Removes the entry at a given position from this list.
    @pre  None.
    @post  If 1 <= position <= getLength() and the removal is successful,
       the entry at the given position in the list is removed, other
       items are renumbered accordingly, and the returned value is true.
    @param position  The list position of the entry to remove.
    @return  True if removal is successful, or false if not. */
   virtual bool remove(int position) = 0;

   /** Removes all entries from this list.
    @post  List contains no entries and the count of items is 0. */
   virtual void clear() = 0;

   /** Gets the entry at the given position in this list.
    @pre  1 <= position <= getLength().
    @post  The desired entry has been returned.
    @param position  The list position of the desired entry.
    @return  The entry at the given position. */
   virtual ItemType getEntry(int position) const = 0;

   /** Replaces the entry at the given position in this list.
    @pre  1 <= position <= getLength().
    @post  The entry at the given position is newEntry.
    @param position  The list position of the entry to replace.
    @param newEntry  The replacement entry. */
   virtual void setEntry(int position, const ItemType& newEntry) = 0;
}; // end ListInterface
```

SUMMARY

1. The ADT list maintains its data by position. Each entry in a list is identified by its position, which is given by an integer, beginning with 1. Thus, the data in a list has an order, but that order is determined by the list's client, not the list itself.
2. You can insert a new entry into a list at a position that ranges from 1 to 1 greater than the current length of the list. Thus, you can insert a new entry before the first entry, after the last entry, or between two current entries.
3. Inserting a new entry into a list renumbers any existing entries that follow the new one in the list.
4. You can remove an entry that is currently at a position that ranges from 1 to the current length of the list. Thus, you can remove the first entry, the last entry, or any interior entry.
5. Removing an entry from a list renumbers any existing entries that follow the deleted one in the list.

Chapter 9: List Implementations

Tuesday, February 23, 2021 2:38 PM

In the last chapter, we specified the ADT List operations - now we are looking at different ways to implement the ADT List, array-based or link-based.

9.1 An Array-Based Implementation of the ADT List

An array-based implementation is a natural choice because both an array and a list identify their items by number. However, the ADT list has operations such as getLength() that the array does not.

- With array-based implementation, you can store a list's *k*th entry in *items[k-1]*.
- You need to keep track of the number of current items in the list.
- The maximum length of the array is a known, fixed value such as *maxItems*.

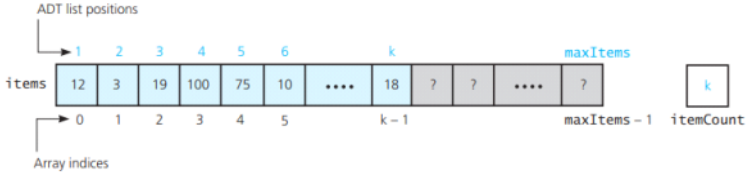
1. **THE HEADER FILE**
   - Derive *ArrayList* class from the template interface *ListInterface*.
   - Provide a default constructor; since we plan to use a statically allocated array, compiler-generated destructor and copy constructor will be sufficient.
   - Throw exceptions for *getEntry* and *setEntry* since the behavior is not specified if the caller violates the precondition.

   - Private data members:

```
static const int DEFAULT_CAPACITY = 100;
ItemType items[DEFAULT_CAPACITY];      // Array of list items
int itemCount;                         // Current count of list items
int maxItems;                          // Maximum capacity of the list
```

   - Operations:

```
bool isEmpty() const;
int getLength() const;
bool insert(int newPosition, const ItemType& newEntry);
bool remove(int position);
void clear();

/** @throw PrecondViolatedExcep if position < 1 or
                               position > getLength(). */
ItemType getEntry(int position) const throw(PrecondViolatedExcep);

/** @throw PrecondViolatedExcep if position < 1 or
                               position > getLength(). */
void setEntry(int position, const ItemType& newEntry)
                            throw(PrecondViolatedExcep);
```



2. **THE IMPLEMENTATION FILE**
   - Default constructor: Initializes data members

```
template<class ItemType>
ArrayList<ItemType>::ArrayList() : itemCount(0),
                                   maxItems(DEFAULT_CAPACITY)
{
} // end default constructor
```

   - *isEmpty:* checks if list is empty

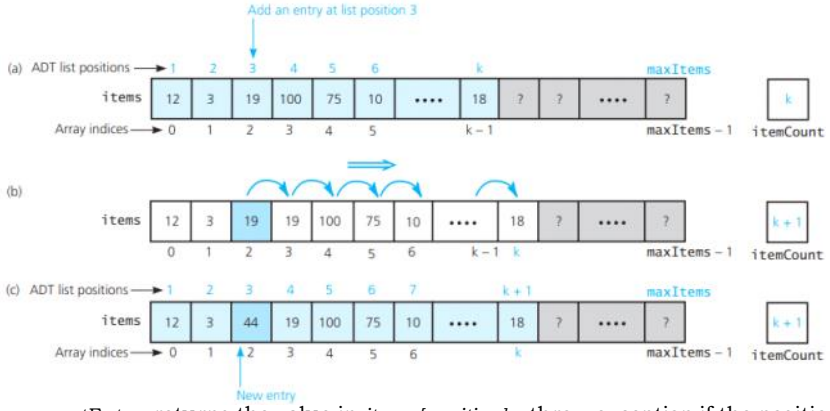
```
template<class ItemType>
bool ArrayList<ItemType>::isEmpty() const
{
   return itemCount == 0;
} // end isEmpty
```

   - *getLength:* get number of items in list

```
template<class ItemType>
int ArrayList<ItemType>::getLength() const
{
   return itemCount;
} // end getLength
```

   - *insert:* shift array entries to insert an item at a given position

```
template<class ItemType>
bool ArrayList<ItemType>::insert(int newPosition,
                                 const ItemType& newEntry)
{
  bool ableToInsert = (newPosition >= 1) &&
                      (newPosition <= itemCount + 1) &&
                      (itemCount < maxItems);
  if (ableToInsert)
  {
    // Make room for new entry by shifting all entries at
    // positions >= newPosition toward the end of the array
    // (no shift if newPosition == itemCount + 1)
    for (int pos = itemCount; pos >= newPosition; pos--)
      items[pos] = items[pos - 1];

    // Insert new entry
    items[newPosition - 1] = newEntry;
    itemCount++;  // Increase count of entries
  } // end if

  return ableToInsert;
} // end insert
```



- *getEntry:* returns the value in *items[position]* - throw exception if the position is out of bounds

```
template<class ItemType>
ItemType ArrayList<ItemType>::getEntry(int position) const
                              throw(PrecondViolatedExcep)
{
  // Enforce precondition
  bool ableToGet = (position >= 1) && (position <= itemCount);
  if (ableToGet)
    return items[position - 1];
  else
  {
    string message = "getEntry() called with an empty list or ";
    message = message + "invalid position.";
    throw(PrecondViolatedExcep(message));
  } // end if
} // end getEntry
```

- *setEntry* (because the client of the class cannot access the class' private members directly)*:* set the item at a given position

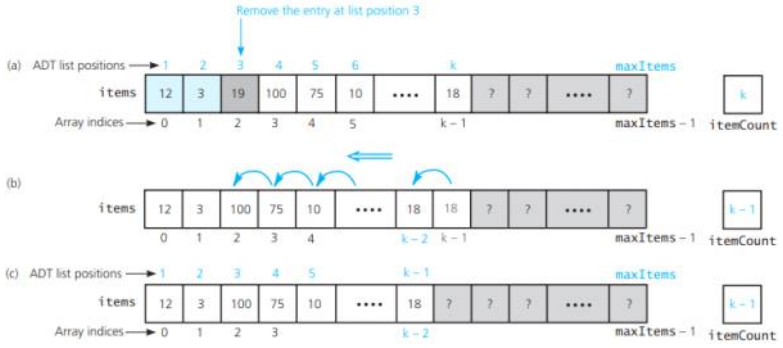
```
template<class ItemType>
void ArrayList<ItemType>::setEntry(int position, const ItemType& newEntry)
                          throw(PrecondViolatedExcep)
{
  // Enforce precondition
  bool ableToSet = (position >= 1) && (position <= itemCount);
  if (ableToSet)
    items[position - 1] = newEntry;
  else
  {
    string message = "setEntry() called with an empty list or ";
    message = message + "invalid position.";
    throw(PrecondViolatedExcep(message));
  } // end if
} // end setEntry
```

- *remove:* removes the item at a given position; however, simply blanking out the item can lead to gaps in the area - instead, you must shift the entries in the array so that a deletion does not leave a gap.

  Problems with having gaps in the array:
  - *itemCount-1* is no longer the index of the last entry in the array. You need another variable, *lastPosition,* to contain this index.
  - Because the items are spread out, *getEntry* will have to look at every cell of the array since *position* is no longer applicable due to the gaps.
  - When *items[maxItems-1]* is occupied, the list could appear full even though it isn't

```
template<class ItemType>
bool ArrayList<ItemType>::remove(int position)
{
```

```
template<class ItemType>
bool ArrayList<ItemType>::remove(int position)
{
   bool ableToRemove = (position >= 1) && (position <= itemCount);
   if (ableToRemove)
   {
      // Remove entry by shifting all entries after the one at
      // position toward the beginning of the array
      // (no shift if position == itemCount)
      for (int fromIndex = position, toIndex = fromIndex - 1;
               fromIndex < itemCount; fromIndex++, toIndex++)
        items[toIndex] = items[fromIndex];

      itemCount--;  // Decrease count of entries
   }  // end if

   return ableToRemove;
}  // end remove
```



- *clear:* clears all items in the list by setting *itemCount* to zero

```
template<class ItemType>
void ArrayList<ItemType>::clear()
{
   itemCount = 0;
}  // end clear
```

Because the memory is statically-allocated, we don't have to worry about actually deleting the items themselves in the memory, right? We just set itemCount to zero so that to the client everything appears to be zero. However, for a link-based implementation, the nodes are allocated dynamically so you have to physically go back and deallocate the memory - you can't just make it appear to be zero.

9.2 A Link-Based Implementation of the ADT List

A link-based implementation does not shift items during insertion and deletion operations. Also does not impose a fixed maximum length on the list. *headPtr* and *itemCount* are the private data members of our class.

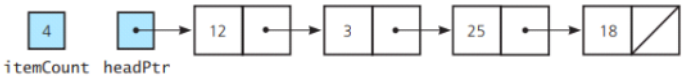
1. **THE HEADER FILE**
   - Need to write your own copy and destructor methods (because dynamically allocated memory)
   - Private data members:

```
Node<ItemType>* headPtr; // Pointer to first node in the chain
                         // (contains the first entry in the list)
int itemCount;           // Current count of list items
```

   - Operations:

```
LinkedList();
LinkedList(const LinkedList<ItemType>& aList);
virtual ~LinkedList();

bool isEmpty() const;
int getLength() const;
bool insert(int newPosition, const ItemType& newEntry);
bool remove(int position);
void clear();

/** @throw  PrecondViolatedExcep if position < 1 or
                                 position > getLength(). */
ItemType getEntry(int position) const throw(PrecondViolatedExcep);

/** @throw PrecondViolatedExcep if position < 1 or
                                 position > getLength(). */
void setEntry(int position, const ItemType& newEntry)
                            throw(PrecondViolatedExcep);
```

     - *getNodeAt* is NOT an ADT operation because it returns a pointer to a node - it is a private method that only the implementations of the ADT's operations call



2. THE IMPLEMENTATION FILE

- Default constructor (will also need a copy constructor): initialize *headPtr* and *itemCount*

```
template<class ItemType>
LinkedList<ItemType>::LinkedList() : headPtr(nullptr), itemCount(0)
{
} // end default constructor
```

- *getEntry*: will get an entry from the list - throws an exception if position is out of bounds

```
template<class ItemType>
ItemType LinkedList<ItemType>::getEntry(int position) const
                                throw(PrecondViolatedExcep)
{
   // Enforce precondition
   bool ableToGet = (position >= 1) && (position <= itemCount);
   if (ableToGet)
   {
      Node<ItemType>* nodePtr = getNodeAt(position);
      return nodePtr->getItem();
   }
   else
   {
      string message = "getEntry() called with an empty list or ";
      message = message + "invalid position.";
      throw(PrecondViolatedExcep(message));
   } // end if
} // end getEntry
```

- *getNodeAt*: locates the node at a given position by traversing the chain and returns the pointer to the located node

```
template<class ItemType>
Node<ItemType>* LinkedList<ItemType>::getNodeAt(int position) const
{
   // Debugging check of precondition
   assert( (position >= 1) && (position <= itemCount) );

   // Count from the beginning of the chain
   Node<ItemType>* curPtr = headPtr;
   for (int skip = 1; skip < position; skip++)
     curPtr = curPtr->getNext();

   return curPtr ;
} // end getNodeAt
```

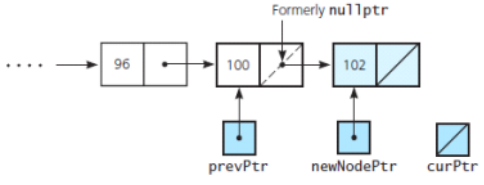
- *Insert*: insert a node between two other nodes

Steps to insert node:

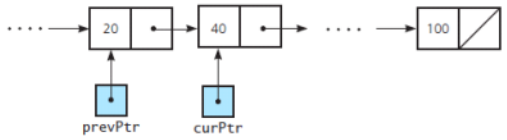
1. Create a new node and store the new data in it
2. Determine the point of insertion
3. Connect the new node to the linked chain by changing pointers

Inserting a new node at the end of the chain is NOT a special case, since *curPtr* becomes *nullPtr* if it moves past the end of the chain. If *curPtr* has the value *nullPtr* and *prevPtr* points to the last node in the chain, this will automatically insert the new node to the end of the list:

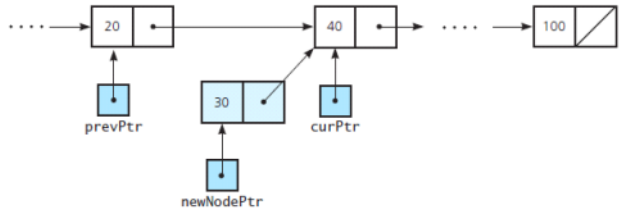
```
newNodePtr->setNext(curPtr);
prevPtr->setNext(newNodePtr);
```



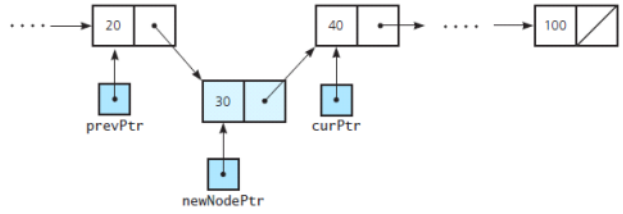
(a) Before the insertion of a new node



(b) After newNodePtr->setNext(curPtr) executes



(c) After prevPtr->setNext(newNodePtr) executes




```
template<class ItemType>
bool LinkedList<ItemType>::insert(int newPosition,
                                  const ItemType& newEntry)
{
  bool ableToInsert = (newPosition >= 1) &&
                      (newPosition <= itemCount + 1);
  if (ableToInsert)
  {
     // Create a new node containing the new entry
     Node<ItemType>* newNodePtr = new Node<ItemType>(newEntry);

     // Attach new node to chain
     if (newPosition == 1)
     {
        // Insert new node at beginning of chain
        newNodePtr->setNext(headPtr);
        headPtr = newNodePtr;
     }
     else
     {
        // Find node that will be before new node
        Node<ItemType>* prevPtr = getNodeAt(newPosition - 1);

        // Insert new node after node to which prevPtr points
        newNodePtr->setNext(prevPtr->getNext());
        prevPtr->setNext(newNodePtr);
     }  // end if

     itemCount++;  // Increase count of entries
  }  // end if

  return ableToInsert;
}  // end insert
```

- *remove*: must be able to delete any node

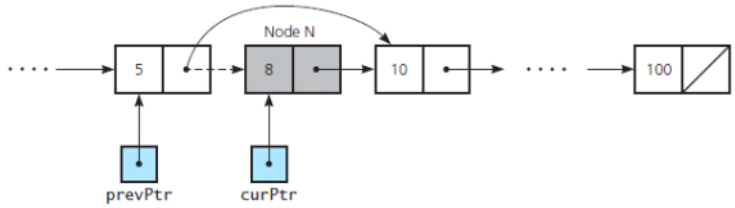
  How to delete an interior node (NOT just the first node):

1. Locate the node you want to delete
2. Disconnect this node from the linked chain by changing pointers
3. Return the node to the system

Assignment statement to remove the node that *curPtr* points to:

```
prevPtr->setNext(curPtr->getNext());
```

- The assignment above works for removing the last node, but NOT the first node - therefore, the first node is a special case.




```
template<class ItemType>
bool LinkedList<ItemType>::remove(int position)
{
   bool ableToRemove = (position >= 1) && (position <= itemCount);
   if (ableToRemove)
   {
      Node<ItemType>* curPtr = nullptr;
      if (position == 1)
      {
         // Remove the first node in the chain
         curPtr = headPtr; // Save pointer to node
         headPtr = headPtr->getNext();
      }
      else
      {
         // Find node that is before the one to delete
         Node<ItemType>* prevPtr = getNodeAt(position - 1);

         // Point to node to delete
         curPtr = prevPtr->getNext();

         // Disconnect indicated node from chain by connecting the
         // prior node with the one after
         prevPtr->setNext(curPtr->getNext());
      }  // end if

      // Return node to system
      curPtr->setNext(nullptr);

      delete curPtr;
      curPtr = nullptr;

      itemCount--;  // Decrease count of entries
   }   // end if

   return ableToRemove;
}  // end remove
```

? More info on how deleting pointers work:

https://stackoverflow.com/questions/13223399/deleting-a-pointer-in-c

- *clear*: to clear the list, simple invoke *remove(1)* until the list is empty

```
template<class ItemType>
void LinkedList<ItemType>::clear()
{
   while (!isEmpty())
      remove(1);
}  // end clear
```

- Destructor: because *clear* invokes *remove* repeatedly, it already takes care of deallocating all the nodes. Therefore, the destructor can simple call *clear*.

```
template<class ItemType>
LinkedList<ItemType>::~LinkedList()
{
   clear();
}  // end destructor
```

3. USING RECURSION IN LinkedList METHODS

- Basic logic for adding a new node to a chain of linked nodes:

```
if (the insertion position is 1)
   Add the new node to the beginning of the chain
else
   Ignore the first node and add the new node to the rest of the chain
```

- Adding to the beginning of the chain. *newNodePtr* points to the new node and *subChainPtr* initially points to the chain and later points to the rest of the chain:

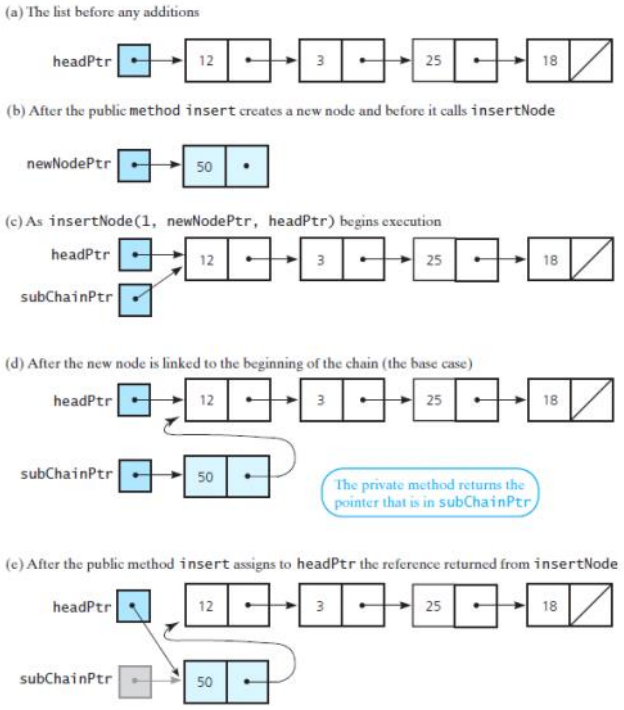
```
if (position == 1)
{
   newNodePtr->setNext(subChainPtr)
   subChainPtr = newNodePtr
   Increment itemCount
}
else
   Using recursion, add the new node at position position - 1 of the subchain pointed
     to by subChainPtr->getNext()
```

---

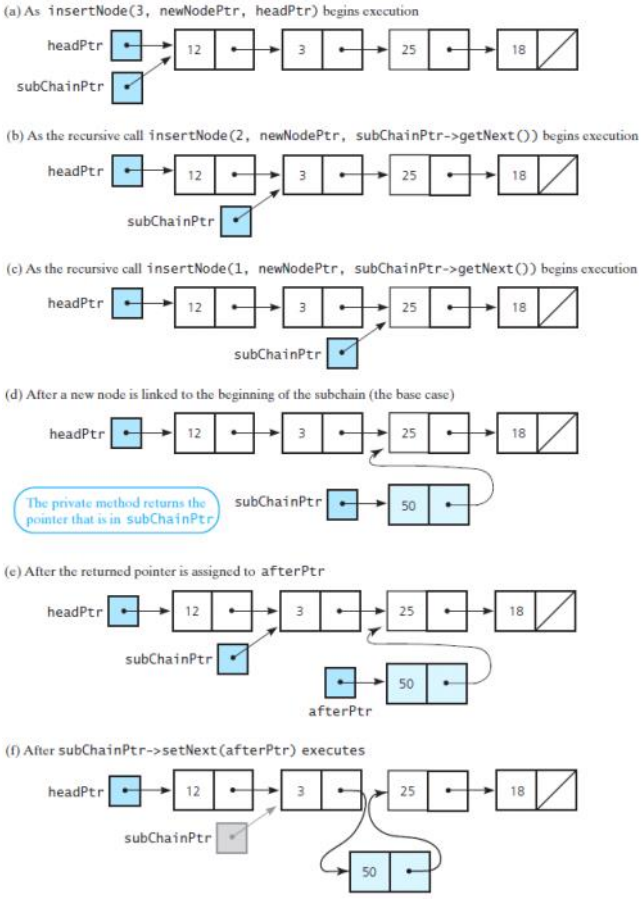
insertNode method:

```
template<class ItemType>
Node<ItemType>* LinkedList<ItemType>::insertNode(int position,
                Node<ItemType>* newNodePtr, Node<ItemType>* subChainPtr)
{
   if (position == 1)
   {
      // Insert new node at beginning of subchain
      newNodePtr->setNext(subChainPtr);
      subChainPtr = newNodePtr;
      itemCount++;  // Increase count of entries
   }
   else
   {
      Node<ItemType>* afterPtr =
          insertNode(position - 1, newNodePtr, subChainPtr->getNext());
      subChainPtr->setNext(afterPtr);
   }  // end if
   return subChainPtr;
}  // end insertNode
```

- Add new node to the beginning of the chain (no recursion necessary):



- Recursively add new node to the interior of the chain:



A recursive addition to a chain of nodes locates and remembers the nodes prior to the insertion point. After the portion of the chain that follows the insertion point is linked to the new node, the recursion links the remembered nodes back into the chain.

9.3 Comparing Implementations

- The time to access the ith node in a chain of linked nodes depends on i.
- You can access array items directly with equal access time.
- Insertions and removals with a link-based implementation do not require you to shift data but require traversal

SUMMARY

1. Using an array results in a straightforward implementation of the ADT list, but it is somewhat more involved than the implementations of either the ADT bag or the ADT stack.
2. An array provides direct access to any of its elements, so a method such as `getEntry` can quickly retrieve any entry in a list.
3. Adding an entry to or removing an entry from an array-based list typically requires that other entries shift by one position within the array. This data movement degrades the time efficiency of these operations, particularly when the list is long and the position of the addition or removal is near the beginning of the list.
4. Inserting entries into or removing entries from a chain of linked nodes does not require data to shift. Thus, the methods `insert` and `remove` of a link-based implementation of the ADT list require essentially the same effort—regardless of the length of the list or the position of the operation within the list—once the point of insertion or removal is known. Finding this point, however, requires a list traversal, the time for which will vary depending on where in the list the operation occurs.
5. For a link-based implementation of the ADT list, adding to the beginning of the list and removing a list's first entry are treated as special cases.
6. Adding or removing an entry at the end of a link-based list requires a traversal of the underlying chain.
7. Adding or removing an entry anywhere within a link-based list requires a change of at most two pointers within the underlying chain.
8. The array-based `getEntry` method is almost instantaneous regardless of which list item you access. A link-based `getEntry`, however, requires i steps to access the i^{th} item in the list.

Chapter 10: Algorithm Efficiency

Tuesday, February 23, 2021 2:38 PM

10.1 What Is a Good Solution?

1. *"A solution is good if the total cost it incurs over all phases of its life is minimal."*
2. Efficiency is only one aspect of a solution's cost
3. Faster is not necessarily better; the following also matters:
   a. The cost of human time to develop and maintain the algorithm
   b. The cost of program execution as measured by the amount of computer time and memory that the program requires to execute
4. The choice of a solution's components -- the objects and the design of the interaction between those objects -- rather than the code you write, has the most significant impact on efficiency.

10.2 Measuring the Efficiency of Algorithms

- Consider efficiency when selecting an algorithm
- A comparison of algorithms should focus on significant differences in efficiency
- Three difficulties with comparing running time of programs instead of algorithms:
  - *How are the algorithms coded?*
    - One algorithm running faster than another could just be the result of better programming - therefore, you are not comparing the algorithms themselves and instead are comparing the *implementations* of the algorithms, which is not what we want.
  - *What computer should you use?*
    - The particular operations required by the algorithms may cause one algorithm to run faster than another on a certain computer. This messes up the search for efficiency.
  - *What data should the programs use?*
    - One algorithm might run uncharacteristically fast with a particular case of data, giving the wrong impression that it is the more efficient one.

- Instead of analyzing the algorithm based on implementation-dependent things, computer scientists use mathematical technique to analyze algorithms independently.

1. THE EXECUTION TIME OF ALGORITHMS

- Counting an algorithm's operations is a way to assess its efficiency
- The following example shows how to analyze an algorithm that traverses a chain of linked nodes and displays the elements:

```
Node<ItemType>* curPtr = headPtr;        ← 1 assignment
while (curPtr != nullptr)                ← n + 1 comparisons
{
   cout << curPtr->getItem() < endl;     ← n writes
   curPtr = curPtr->getNext();           ← n assignments
}  // end while
```

- If we have n nodes, these statements require $n + 1$ assignments (step 1 + step 4), $n + 1$ comparisons (step 2, since it has to check one more time to decide whether to execute or not), and n write operations (step 4). If each assignment, comparison, and write operation requires, respectively, a, c, and w time units, the statements require $(n + 1) * (a + c) + n * w$ time units. Therefore, the time required to write n nodes is proportional to n.

- For the Towers of Hanoi, the number of moves required for n disks is $2^n - 1$ movies. If each movie requires the same time m, the solution requires $(2^n - 1) * m$ time units. Time requirement increases rapidly as the number of disks increases.

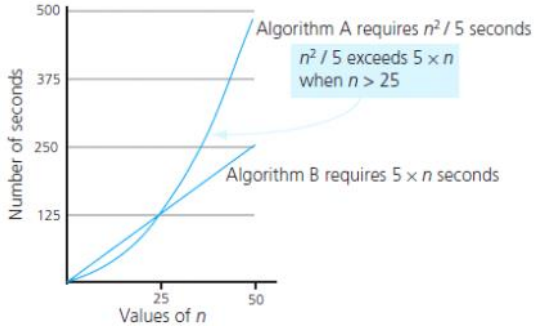
2. ALGORITHM GROWTH RATES

- Measure an algorithm's time requirement as a function of the problem size.
- Do NOT give specifics such as the following line, since it depends on the data used and the implementation of the algorithm:

Algorithm A requires $n^2 / 5$ time units to solve a problem of size n
Algorithm B requires $5 \times n$ time units to solve a problem of size n

- Instead, the most important thing to know is how quickly the algorithm's time requirement grows as a function of the problem size.
- Compare algorithms' growth rates using statements such as:

Algorithm A requires time proportional to n^2
Algorithm B requires time proportional to n



- Algorithm efficiency is typically a concern for large problems only, so even though Algorithm A requires less time for $n < 25$, it doesn't matter because we assume large values of n when analyzing algorithms.

3. ANALYSIS AND BIG O NOTATION

Note: Definition of the order of an algorithm

Algorithm A is order $f(n)$—denoted $O(f(n))$—if constants k and n_0 exist such that A requires no more than $k \times f(n)$ time units to solve a problem of size $n \geq n_0$.

- Growth-rate function of Algorithm A: *f(n)*
- Because the notation uses the capital letter O to denote *order*, it is called the Big O notation.
- Ex: If a problem of size n requires time that is directly proportional to n, the problem is O*(n)*.

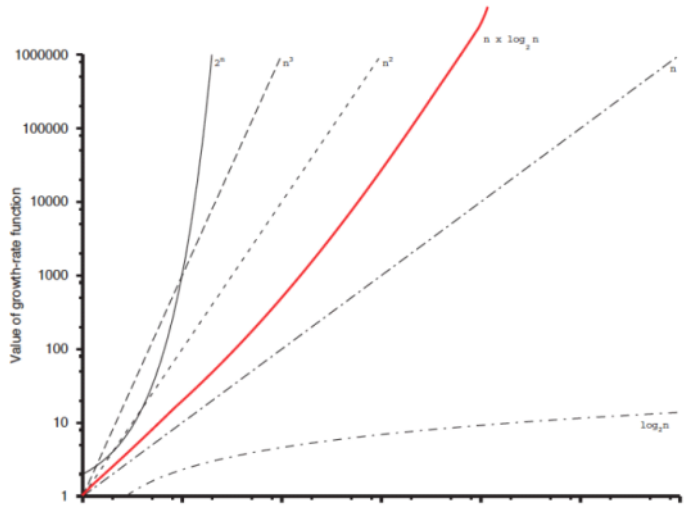
Still super confused about the examples in the textbook…

- Properties of growth-rate functions:
  - You can ignore low-order terms in an algorithms growth-rate function
  - You can ignore a multiplicative constant in the high-order term of an algorithm's growth-rate function
    - For example, O(5*n^3) --> O(n^3)
  - O(*f(n)*) + O(*g(n)*) = O(*f(n)* + *g(n)*)
- An algorithm can require different times to solve different problems of the same size, which is why there are such things as worst-case and average-case analyses. Worst-case does not happen often - instead it shows that the algorithm is never slower than your estimate.
- Average-case analysis attempts to determine the average amount of time that an algorithm requires to solve problems of size n. It is much more difficult to determine that worst-case analysis so worst-case analysis is more common.

More info on Big O Notation:

Big-O notation in 5 minutes — The basics

- Drop low-order terms
- Ignore constants



4. KEEPING YOUR PERSPECTIVE

- An array-based *getEntry* is O(1)
- A link-based *getEntry* is O*(n)*
- We are interested in only significant differences in efficiency
- When choosing an implementation of an ADT, consider how frequently particular ADT operations occur in a given application
- Some seldom-used but critical operations must be efficient
- If the problem size is always small, you can probably ignore an algorithm's efficiency
- Weigh the trade-offs between an algorithm's time requirements and its memory requirements
- Compare algorithms for both style and efficiency
- Analysis of an algorithm's time efficiency focuses on large problems

5. THE EFFICIENCY OF SEARCHING ALGORITHMS

- Sequential search
  - Worst-case: O(*n*)
  - Average-case: O(*n*)
  - Best-case: O(1)

- Binary search
  - Worst-case: O(*log2(n)*)

- NOTE: If you are searching a very large array, an O(*n*) algorithm is probably too inefficient to use.

SUMMARY

1. Using Big O notation, you measure an algorithm's time requirement as a function of the problem size by using a growth-rate function. This approach enables you to analyze the efficiency of an algorithm without regard for such factors as computer speed and programming skill that are beyond your control.
2. When you compare the inherent efficiency of algorithms, you examine their growth-rate functions when the problems are large. Only significant differences in growth-rate functions are meaningful.
3. Worst-case analysis considers the maximum amount of work an algorithm requires on a problem of a given size, while average-case analysis considers the expected amount of work that it requires.
4. Analyzing an algorithm's time requirement will help you to choose an implementation for an abstract data type. If your application frequently uses particular ADT operations, your implementation should be efficient for at least those operations.

Chapter 12: Sorted Lists & Their Implementations

Tuesday, February 23, 2021 2:38 PM

The ADT sorted list maintains its entries in sorted order.

12.1 Specifying the ADT Sorted List

- The ADT sorted list differs from the ADT list in that a sorted list inserts and removes items by their values and not by their positions. This is because the sorted list insertion operation determines the position of the new entry based on its value - therefore, there is no operation to insert entries by position since it might destroy the order of the sorted list's entries.

ABSTRACT DATA TYPE: SORTED LIST

DATA

- A finite number of objects, not necessarily distinct, having the same data type and ordered by their values.

OPERATIONS

PSEUDOCODE	DESCRIPTION
`insertSorted(newEntry)`	Task: Inserts an entry into this sorted list in its proper order so that the list remains sorted. Input: `newEntry` is the new entry. Output: None.
`removeSorted(anEntry)`	Task: Removes the first or only occurrence of `anEntry` from this sorted list. Input: `anEntry` is the entry to remove. Output: Returns true if `anEntry` was located and removed, or false if not. In the latter case, the list remains unchanged.
`getPosition(anEntry)`	Task: Gets the position of the first or only occurrence of `anEntry` in this sorted list. Input: `anEntry` is the entry to be located. Output: Returns the position of `anEntry` if it occurs in the sorted list. Otherwise, returns the position where `anEntry` would occur in the list, but as a negative integer.

The following operations behave as they do for the ADT list and are described in Chapter 8:

```
isEmpty()
getLength()
remove(position)
clear()
getEntry(position)
```

- Example of the ADT sorted list operations in use:

```
nameListPtr->insertSorted("Sarah");
nameListPtr->insertSorted("Tom");
nameListPtr->insertSorted("Carlos");
```

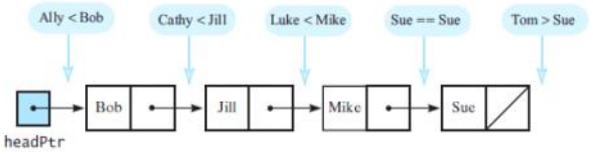
The sorted list now contains the following entries:

Brenda
Carlos
Jamie
Sarah
Tom

- Basically, the ADT sorted list can add, remove, or locate an entry, given the entry as an argument.

12.2 A Link-Based Implementation

- Very similar to the link-based implementation of the ADT list from Chapter 9.
- The class *LinkedSortedList* requires three new public methods compared to the un-sorted list (each method takes an entry as an argument):
  - *insertSorted()*
  - *removeSorted()*
  - *getPosition()*
- How *insertSorted* will work:



- Pseudocode to add a new entry to the sorted list:

```
// Adds a new entry to the sorted list.
add(newEntry)

   Allocate a new node containing newEntry
   Search the chain until either you find a node containing newEntry or you pass
      the point where it should be
   Let prevPtr point to the node before the insertion point
   if (the chain is empty or the new node belongs at the beginning of the chain)
      Add the new node to the beginning of the chain
   else
      Insert the new node after the node referenced by prevPtr

   Increment the length of the sorted list
```

- The performance of *insertSorted* depends on the efficiency of the method *getNodeBefore*, which locates the insertion point by traversing the chain of nodes. This traversal is O *(n)*, making addition to a sorted list and O *(n)* operations.

12.3 Implementations That Use the ADT List

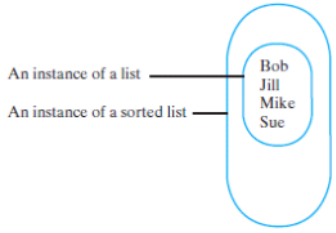
Containment vs. Inheritance

Advanced C++: Code Reuse - Inheritance vs Composition

- Because the ADT sorted lists overlaps a lot with the ADT list, we can use the ADT list when implementing the ADT sorted list. This can be done using one of three techniques:
  - Containment
  - Public inheritance
  - Private inheritance

Just check the textbook for more detailed info on the implementation…at this point, the words are just going in one eye and out the other - nothing is registering…

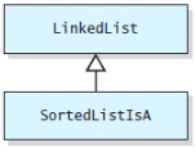
CONTAINMENT



- This containment type is composition and illustrates the has-a relationship between the class of sorted lists and the class of lists.

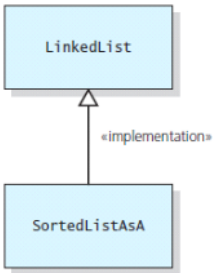
ADT Sorted List Operation	List Implementation	
	Array-based	Link-based
`insertSorted(newEntry)`	$O(n)$	$O(n^2)$
`removeSorted(anEntry)`	$O(n)$	$O(n^2)$
`getPosition(anEntry)`	$O(n)$	$O(n^2)$
`getEntry(position)`	$O(1)$	$O(n)$
`remove(givenPosition)`	$O(n)$	$O(n)$
`clear()`	$O(1)$	$O(n)$
`getLength(), isEmpty()`	$O(1)$	$O(1)$

PUBLIC INHERITANCE



- An is-a relationship implies public inheritance
- The sorted list class is a descendant of LinkedList
- Efficiency is similar to that of containment technique

PRIVATE INHERITANCE



- If you want to inherit members from the existing class, you can use private inheritance. Private inheritance enables you to use the methods of a base class without giving a client access to them.

Summary

1. The ADT sorted list maintains its entries in sorted order. It, not the client, determines where to place an entry.
2. The ADT sorted list can add, remove, or locate an entry, given the entry as an argument.
3. The ADT sorted list has several operations that are the same as the corresponding operations of the ADT list. However, a sorted list will not let you add or replace an entry by position.
4. A chain of linked nodes provides a reasonably efficient implementation of the sorted list.
5. A class of sorted lists that has a list as a data field is said to use containment. Although such a class is easy to write, its efficiency can suffer if the implementation of the ADT list is inefficient.
6. Although it seems like a sorted list is a list, deriving a class of sorted lists from a class of lists using public inheritance is not appropriate. Doing so requires you to override some methods of the ADT list that a client of the sorted list class should not be able to use. Additionally, public inheritance would make the two classes object-type compatible, but you cannot use a sorted list anywhere that a list is used.
7. Private inheritance provides a reasonable alternative to containment as an approach to using the ADT list in the implementation of the ADT sorted list. Usually, however, containment is preferable.

Important Tags

Sunday, April 4, 2021 9:25 PM

Important

- *A recursive addition to a chain of nodes locates and remembers the nodes prior*
- Algorithm efficiency is typically a concern for large problems only, so even though Algorithm A requires less time for $n < 25$, it doesn't matter because we assume large values of n when analyzing algorithms.
- Basically, the ADT sorted list can add, remove, or locate an entry, given the entry as an argument.
- *Because the memory is statically-allocated, we don't have to worry about actually deleting the items themselves in the memory, right? We just set itemCount to zero so that to the client everything appears to be zero. However, for a link-based implementation, the nodes are allocated dynamically so you have to physically go back and deallocate the memory - you can't just make it appear to be zero.*
- *getNodeAt* is NOT an ADT operation because it returns a pointer to a node - it is a private method that only the implementations of the ADT's operations call
- If a shallow copy of the stack is sufficient, you can omit the copy constructor, in which case the compiler generates a copy constructor that performs a shallow copy.
- Instead, the most important thing to know is how quickly the algorithm's time requirement grows as a function of the problem size.
- NOTE: If you are searching a very large array, an O(*n*) algorithm is probably too inefficient to use.
- The assignment above works for removing the last node, but NOT the first node - therefore, the first node is a special case.
- The performance of *insertSorted* depends on the efficiency of the method *getNodeBefore*, which locates the insertion point by traversing the chain of nodes. This traversal is O*(n)*, making addition to a sorted list and O*(n)* operations.
- Your use of an ADT's operations should note depend on its implementation

Question

- Containment vs. Inheritance
- More info on Big O Notation:
- To clarify difference between abstraction and implementation:
- What is **assert()** in C++?
- What is **statically allocated memory** and **dynamically allocated memory?**