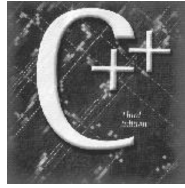
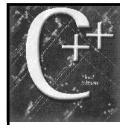


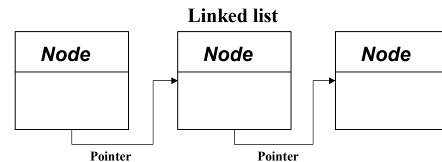
Pointers & Dynamic Data Structures

Chapter 13



Dynamic Data Structures

- Arrays & structs are static (compile time)
- Dynamic expand as program executes
- Linked list is example of dynamic data structure



13.1 Pointers and the "new" Operator

- Pointer Declarations
  - pointer variable of type "pointer to float"
  - can store the address of a float in p

    float *p;

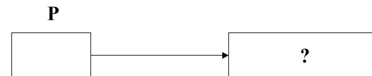
- The new operator creates a variable of type float & puts the address of the variable in pointer p

    p = new float;

- Dynamic allocation - program execution

Pointers

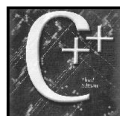
- Actual address has no meaning



- Form: *type *variable;*
- Example: *float *p;*

new Operator

- Actually allocates storage
- Form: ***new type;***
  new type [n];
- Example: ***new float;***

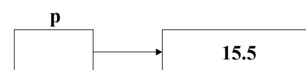


Accessing Data with Pointers

- * - indirection operator

    ****p = 15.5;***

- Stores floating value 15.5 in memory location *p - the location pointed to by p



Pointer Statements

```
float   *p;
p = new float;
*p = 15.5;
cout << “The contents of the memory cell pointed to
                        by p is “ << *p << endl;
```

Output

The contents of memory cell pointed to by p is 15.5

Pointer Operations

- Pointers can only contain addresses
- So the following are errors:
  - ***p = 1000;***
  - ***p = 15.5;***
- Assignment of pointers if q & p are the same pointer type
  - ***q = p;***
- Also relational operations == and ***!=***

Pointers to Structs

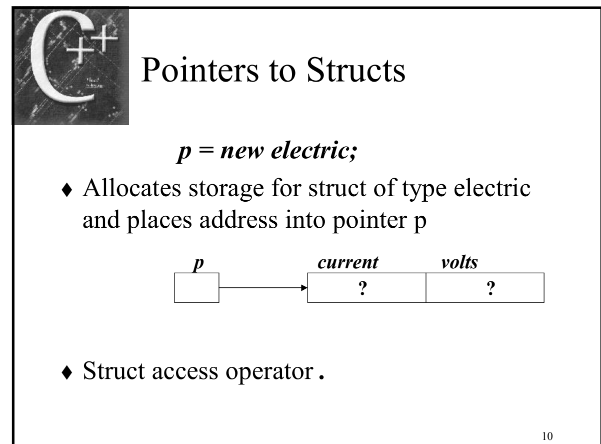
```
struct electric
{
    string current;
    int volts;
};
electric     *p,     *q;
```

- p and q are pointers to a struct of type electric

Pointers to Structs

p = new electric;

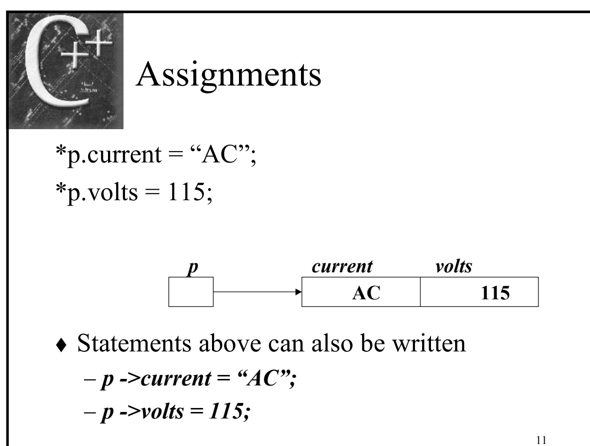
- Allocates storage for struct of type electric and places address into pointer p



- Struct access operator **.**

Assignments

```
*p.current = “AC”;
*p.volts = 115;
```



- Statements above can also be written
  - ***p ->current = “AC”;***
  - ***p ->volts = 115;***

Struct Member Access via Pointers

- From: ***p ->m***
- Example: ***p ->volts***
- ***Example:***
  - ***cout << p->current << p->volts << endl;***
- Output
  - AC115

Pointers to Structs

q = new electric;

- Allocates storage for struct of type electric and places address into pointer q
- Copy contents of p struct to q struct

q = *p;

q | q->current | q->volts
AC | 115

Pointers to Structs

q->volts = 220;

q | q->current | q->volts
AC | 220

q = p; p->current | p->volts
p | q->current | q->volts
AC | 115
q
AC | 220

13.2 Manipulating the Heap

- When ***new*** executes where is struct stored ?
- Heap
  - C++ storage pool available to ***new*** operator
- Effect of ***p = new node;***
- Figure 14.1 shows Heap before and after executing new operator

Effect on new on the Heap

Returning Cells to the Heap

- Operation
  - ***delete p;***
- Returns cells back to heap for re-use
- When finished with a pointer delete it
- Watch dual assignments and initialization

- Form: ***delete variable;***
- Example: ***delete p;***

13.3 Linked Lists

- Arrange dynamically allocated structures into a new structure called a ***linked list***
- Think of a set of children's pop beads
- Connecting beads to make a chain
- You can move things around and re-connect the chain
- We use pointers to create the same effect

Children's Beads

Pop bead

Chain of pop beads

Figure 13.2 1R
Friedman/Koffman
Addison Wesley Pearson

Delgado Design, Inc.

Declaring Nodes

- If a pointer is included in a struct we can connect nodes

```
struct node
{
   string word;
   int count;
   node *link;
};
node     *p, *q, *r;
```


Declaring Nodes

- Each var p, q and r can point to a struct of type node
  - word (string)
  - count (int)
  - link (pointer to a node address)

Struct of type node

word	count	link
String	Integer	Address

Connecting Nodes

- Allocate storage of 2 nodes

```
p = new node;
q = new node;
```

- Assignment Statements

```
p->word = "hat";
p->count = 2;
q->word = "top";
q->count = 3;
```


Figure 13.3

p	p -> word	p -> count	p -> link
• →	hat	2	?

q	q -> word	q -> count	q -> link
• →	top	3	?

Figure 13.3 1R
Friedman/Koffman
Addison Wesley Pearson

Delgado Design, Inc.

Connecting Nodes

- Link fields undefined until assignment

  p->link = q;

- Address of ***q*** is stored in link field pointed to by ***p***
- Access elements as follows

  q->word or ***p->link->word***

- ***Null*** stored at last link field

  q->link = NULL; or p->link->link = NULL;

Connecting Nodes

Figure 13.4 1R
Friedman/Koffman
Addison Wesley Pearson

Delgado Design, Inc.

Inserting a Node

- Create and initialize node
  r = new node;
  r->word = "the";
  r->count = 5;
- Connect node pointed to by ***p*** to node pointed to by ***r***
  p->link = r;
- Connect node pointed to by ***r*** to node pointed to
- by ***q***
  r->link = q;

Inserting a New Node in a List

Insertion at Head of List

- OldHead points to original list head
  oldHead = p;
- Point ***p*** to a new node
  p = new node;
- Connect new node to old list head
  p->link = oldHead;

Insertion at Head of List

Figure 13.6 1R
Friedman/Koffman
Addison Wesley Pearson

Delgado Design, Inc.

Insertion at End of List

- Typically less efficient (no pointer)
- Attach new node to end of list
  last->link = new node;
- Mark end with a ***NULL***
  last->link->link = NULL;

Insertion at End of List

Figure 13.7 1R
Friedman/Koffman
Addison Wesley Pearson

Delgado Design, Inc.

Deleting a Node

- Adjust the link field to remove a node
- Disconnect the node pointed to by ***r***
  p->link = r->link;
- Disconnect the node pointed to by ***r*** from its successor
  r->link = NULL;
- Return node to Heap
  delete r;

Deleting a Node

Figure 13.8 1R
Friedman/Koffman
Addison Wesley Pearson

Delgado Design, Inc.

Traversing a List

- Often need to traverse a list
- Start at head and move down a trail of pointers
- Typically displaying the various nodes contents as the traversing continues
- Advance node pointer
  head = head->link;
- Watch use of reference parameters

PrintList.cpp

```
// FILE: PrintList.cpp
// DISPLAY THE LIST POINTED TO BY HEAD

void printList (listNode *head)
{
   while (head != NULL)
   {
      cout << head->word << " " << head ->count
                                   << endl;
      head = head->link;
   }
}
```


Circular Lists - Two Way Option

- A list where the last node points back to the first node
- Two way list is a list that contains two pointers
  - pointer to next node
  - pointer to previous node

13.4 Stacks as Linked Lists

- Implement Stack as a dynamic structure
  - Earlier we used arrays (chps 12, 13)
- Use a *linked list*
- The first element is *s.top*
- New nodes are inserted at head of list
- LIFO (Last-In First-Out)
- StackLis.h

StackList.h

```
//FILE: StackList.h

#ifndef STACK_LIST_H
#define STACK_LIST_H

template <class stackElement>
class stackList
{
   public:
      // Member functions ...
      // CONSTRUCTOR TO CREATE AN EMPTY STACK
      stackList ();
```

StackList.h

```
      bool push (const stackElement& x);
      bool pop (stackElement& x);
      bool peek (stackElement& x) const;
      bool isEmpty () const;
      bool isFull () const;
private:
      struct stackNode
      {
         stackElement item;
         stackNode* next;
      };
```

StackList.h

```
      // Data member
      stackNode* top;
};

#endif  // STACK_LIST_H
```

StackList.cpp

```
// File: stackList.cpp
// Implementation of template class stack
// as a linked list

      #include "stackList.h"
      #include <cstdlib>              // for NULL
      using namespace std;
```

StackList.cpp

```
      // Member functions ...
      template <class stackElement>
      stackList<stackElement>::stackList()
      {
         top = NULL;
      }  // end stackList
      // Push an element onto the stack
      // Pre:  The element x is defined.
      // Post: If there is space on the heap,
      // the item is pushed onto the stack and
      // true is returned. Otherwise, the
      // stack is unchanged and false is
      // returned.
```

StackList.cpp

```
template <class stackElement>
bool stackList<stackElement>::push
        (const stackElement& x)
{
         // Local data
         stackNode* oldTop;
         bool success;

         oldTop = top;
         top = new stackNode;
         if (top == NULL)
         {
```

StackList.cpp

```
            top = oldTop;
            success = false;
         }
         else
         {
            top->next = oldTop;
            top->item = x;
            success = true;
         }
         return success;
      }  // end push
```

StackList.cpp

```
// Pop an element off the stack
// Pre:  none
// Post: If the stack is not empty, the value
// at the top of the stack is removed, its
// value is placed in x, and true is returned.
// If the stack is empty, x is not defined and
// false is returned.
template <class stackElement>
bool stackList<stackElement>::pop
        (stackElement& x)
      {
         // Local data
```

StackList.cpp

```
         stackNode* oldTop;
         bool success;
         if (top == NULL)
            success = false;
         else
         {
            x = top->item;
            oldTop = top;
            top = oldTop->next;
            delete oldTop;
            success = true;
         }
```

StackList.cpp

```
         return success;
      }  // end pop


// Get top element from stack without popping
// Pre:  none
// Post: If the stack is not empty, the value
// at the top is copied into x and true is
// returned. If the stack is empty, x is not
// defined and false is returned. In
// either case, the stack is not changed.
```

StackList.cpp

```
template <class stackElement>
bool stackList<stackElement>::peek
        (stackElement& x) const
{
         // Local data
         bool success;
         if (top == NULL)
            success = false;
         else
         {
            x = top->item;
            success = true;
```

StackList.cpp

```
        }
        return success;
    }  // end peek
// Test to see if stack is empty
// Pre : none
// Post: Returns true if the stack is empty;
// otherwise, returns false.
template <class stackElement>
bool stackList<stackElement>::isEmpty() const
{
        return top == NULL;
}  // end isEmpty
```

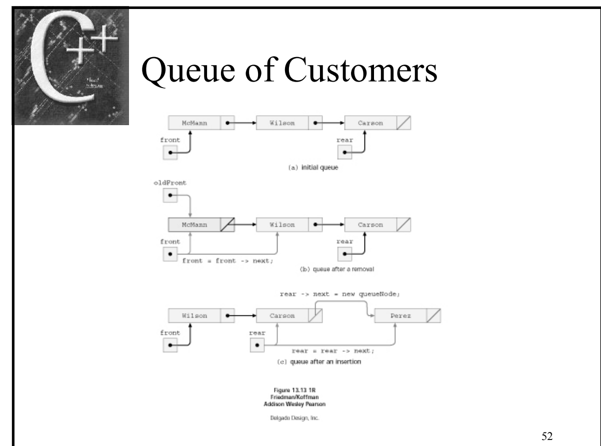

StackList.cpp

```
// Test to see if stack is full
// Pre : none
// Post: Returns false. List stacks are never
// full. (Does not check heap availability.)
template <class stackElement>
bool stackList<stackElement>::isFull() const
{
   return false;
}  // end isFull
```


13.5 Queue ADT

- List structure where items are added to one end and removed from the opposite end
- FIFO (First-In First-Out)
- Bank service line, car wash or check-out are examples of a ***queue***
- Implementing a queue as a list we added elements to the end and remove from the front
- Queue.h

Queue of Customers



Queue.h

```
// FILE: Queue.h
// DEFINITION AND IMPLEMENTATION OF A TEMPLATE
// CLASS QUEUE USING A LINKED LIST

#ifndef QUEUE_H
#define QUEUE_H

template<class queueElement>
class queue
{
   public:
      queue ();
```


Queue.h

```
      bool insert (const queueElement& x);
      bool remove (queueElement& x);
      bool isEmpty ();
      int getSize ();
   private:
      struct queueNode
      {
         queueElement item;
         queueNode* next;
      };
```

Queue.h

```
      queueNode* front;
      queueNode* rear;
      int numItems;
};
#endif   // QUEUE_H
```

Queue.cpp

```
// File: queue.cpp
// Implementation of template class queue

#include "queue.h"
#include <cstdlib>        // for NULL

using namespace std;

// Member functions
// constructor - create an empty queue
template<class queueElement>
queue<queueElement>::queue()
```

Queue.cpp

```
{
     numItems = 0;
     front = NULL;
     rear = NULL;
}
// Insert an element into the queue
// Pre : none
// Post: If heap space is available, the
// value x is inserted at the rear of the queue
// and true is returned. Otherwise, the queue is
// not changed and false is returned.
```

Queue.cpp

```
template<class queueElement>
bool queue<queueElement>::insert
  (const queueElement& x)
      {
         if (numItems == 0)
         {
            rear = new queueNode;
            if (rear == NULL)
               return false;
            else
               front = rear;
         }
```

Queue.cpp

```
         else
         {
            rear->next = new queueNode;
            if (rear->next == NULL)
               return false;
            else
               rear = rear->next;
         }
         rear->item = x;
         numItems++;
         return true;
      }  // end insert
```

Queue.cpp

```
// Remove an element from the queue
// Pre : none
// Post: If the queue is not empty, the value at
// the front of the queue is removed, its value
// is placed in x, and true is returned. If the
// queue is empty, x is not defined and false
// is returned.
template<class queueElement>
bool queue<queueElement>::remove
(queueElement& x)
{
```

Queue.cpp

```
// Local data
queueNode* oldFront;

if (numItems == 0)
{
   return false;
}
else
{
   oldFront = front;
   x = front->item;
   front = front->next;
```


Queue.cpp

```
         oldFront->next = NULL;
         delete oldFront;
         numItems--;
         return true;
      }
   } // end remove
// Test whether queue is empty
template<class queueElement>
bool queue<queueElement>::isEmpty()
{
  return (numItems == 0);
}
```


Queue.cpp

```
// Returns queue size
template<class queueElement>
int queue<queueElement>::getSize()
{
   return numItems;
}
```


13.6 Binary Trees

- List with additional pointer
- 2 pointers
  - right pointer
  - left pointer
- Binary Tree
  - 0 - 1 or 2 successor nodes
  - empty
  - root
  - left and right sub-trees

Binary Tree

FOX DOG HEN CAT ELF HAT HOG (a)

35 25 40 15 30 45 42 (b)

Figure 13.16 1R
Friedman/Koffman
Addison Wesley Pearson

Delgado Design, Inc.

Binary Search Tree

- Efficient data retrieval
- Data stored by ***unique key***
- Each node has 1 data component
- Values stored in right sub-tree are greater than the values stored in the left sub-tree
- Above must be true for all nodes in the binary search tree

Searching Algorithm

if (tree is empty)
  target is not in the tree
else if (the target key is the root)
  target found in root
else if (target key smaller than the root's key)
  search left sub-tree
else
  search right sub-tree

Searching for Key 42

Figure 13.17 1R
Friedman/Koffman
Addison Wesley Pearson

Delgado Design, Inc.

Building a Binary Search Tree

- Tree created from root downward
- Item 1 stored in root
- Next item is attached to left tree if value is smaller or right tree if value is larger
- When inserting an item into existing tree must locate the items parent and then insert

Algorithm for Insertion

if (tree is empty)
  insert new item as root
else if (root key matches item)
  skip insertion duplicate key
else if (new key is smaller than root)
  insert in left sub-tree
else
  insert in right sub-tree

Figure 13.18 Building a Tree

Figure 13.18 1R
Friedman/Koffman
Addison Wesley Pearson

Delgado Design, Inc.

Displaying a Binary Search Tree

- Recursive algorithm

  if (tree is not empty)
    display left sub-tree
    display root
    display right sub-tree
- In-order traversal
- Pre and post order traversals

Example of traversal

- Trace of Figure 13.18
  - Display left sub-tree of node 40
  - Display left sub-tree of node 20
  - Display left sub-tree of node 10
  - Tree is empty - return left sub-tree node is 10
  - ***Display item with key 10***
  - Display right sub-tree of node 10

Example of traversal

- Tree is empty - return from displaying right sub-tree node is 10
- Return from displaying left sub-tree of node 20
- ***Display item with key 20***
- Display right sub-tree of node 20
- Display left sub-tree of node 30
- Tree is empty - return from displaying left sub-tree of node 30
- ***Display item with key 30***

Example of traversal

- Display right sub-tree of node 30
- Tree is empty - return from displaying right sub-tree of node 30
- Return from displaying right sub-tree of node 20
- Return from displaying left sub-tree of node 40
- ***Display item with key 40***
- Display right sub-tree of node 40

13.7 Binary Search Tree ADT

- Specification for a Binary Search Tree
  - root — pointer to the tree root
  - binaryTree — a constructor
  - insert — inserts an item
  - retrieve — retrieves an item
  - search — locates a node for a key
  - display — displays a tree

BinaryTree.h

```
// FILE: BinaryTree.h
// DEFINITION OF TEMPLATE CLASS BINARY SEARCH
// TREE

#ifndef BIN_TREE_H
#define BIN_TREE_H

// Specification of the class
template<class treeElement>
class binTree
{
```


BinaryTree.h

```
public:
   // Member functions ...
   // CREATE AN EMPTY TREE
   binTree ();
   // INSERT AN ELEMENT INTO THE TREE
   bool insert (const treeElement& el );

   // RETRIEVE AN ELEMENT FROM THE TREE
   bool retrieve (treeElement& el ) const;

   // SEARCH FOR AN ELEMENT IN THE TREE
   bool search (const treeElement& el )
                                       const;
```

BinaryTree.h

```
      // DISPLAY A TREE
      void display () const;

    private:
      // Data type ...
       struct treeNode
      {
         treeElement info;
         treeNode* left;
         treeNode* right;
      };
```


BinaryTree.h

```
      // Data member ....
      treeNode* root;

      // Member functions ...
      // Searches a subtree for a key
      bool search (treeNode*, const
                        treeElement&) const;
      // Inserts an item in a subtree
      bool insert (treeNode*&, const
                         treeElement&) const;
      // Retrieves an item in a subtree
      bool retrieve (treeNode*,
                        treeElement&) const;
```


BinaryTree.h

```
      // Displays a subtree
      void display (treeNode*) const;
};

#endif   // BIN_TREE_H
```


BinaryTree.cpp

```
// File: binaryTree.cpp
// Implementation of template class binary
// search tree
#include "binaryTree.h"
#include <iostream>
using namespace std;
// Member functions ...
// constructor - create an empty tree
template<class treeElement>
binaryTree<treeElement>::binaryTree()
{
   root = NULL;
}
```


BinaryTree.cpp

```
// Searches for the item with same key as el
//  in a binary search tree.
// Pre : el is defined.
// Returns true if el's key is located,
//   otherwise, returns false.
template<class treeElement>
bool binaryTree<treeElement>::search
    (const treeElement& el) const
{
   return search(root, el);
} // search
```


BinaryTree.cpp

```
// Searches for the item with same key as el
// in the subtree pointed to by aRoot. Called
// by public search.
// Pre : el and aRoot are defined.
// Returns true if el's key is located,
// otherwise, returns false.
template<class treeElement>
bool binaryTree<treeElement>::search
     (treeNode* aRoot,const treeElement& el)
                              const
{
    if (aRoot == NULL)
```

BinaryTree.cpp

```
      return false;
    else if (el == aRoot->info)
      return true;
    else if (el <= aRoot->info)
      return search(aRoot->left, el);
    else
      return search(aRoot->right, el);
  } // search
```

BinaryTree.cpp

```
// Inserts item el into a binary search tree.
// Pre : el is defined.
// Post: Inserts el if el is not in the tree.
// Returns true if the insertion is performed.
// If there is a node with the same key value
// as el, returns false.
template<class treeElement>
bool binaryTree<treeElement>::insert
  (const treeElement& el)
{
    return insert(root, el);
} // insert
```

BinaryTree.cpp

```
// Inserts item el in the tree pointed to by
// aRoot.
// Called by public insert.
// Pre : aRoot and el are defined.
// Post: If a node with same key as el is found,
// returns false. If an empty tree is reached,
// inserts el as a leaf node and returns true.
template<class treeElement>
bool binaryTree<treeElement>::insert
  (treeNode*& aRoot,
   const treeElement& el)
{
```

BinaryTree.cpp

```
  // Check for empty tree.
  if (aRoot == NULL)
  { // Attach new node
    aRoot = new treeNode;
    aRoot->left = NULL;
    aRoot->right = NULL;
    aRoot->info = el;
    return true;
  }
  else if (el == aRoot->info)
    return false;
```

BinaryTree.cpp

```
  else if (el <= aRoot->info)
    return insert(aRoot->left, el);
  else
    return insert(aRoot->right, el);
} // insert

// Displays a binary search tree in key order.
// Pre : none
// Post: Each element of the tree is displayed.
// Elements are displayed in key order.
```

BinaryTree.cpp

```
template<class treeElement>
void binaryTree<treeElement>::display() const
{
   display(root);
} // display
// Displays the binary search tree pointed to
// by aRoot in key order. Called by display.
// Pre : aRoot is defined.
// Post: displays each node in key order.
template<class treeElement>
void binaryTree<treeElement>::display
  (treeNode* aRoot) const
```

BinaryTree.cpp

```
{
   if (aRoot != NULL)
   { // recursive step
     display(aRoot->left);
     cout << aRoot->info << endl;
     display(aRoot->right);
   }
} // display
```

BinaryTree.cpp

```
// Insert member functions retrieve.
template<class treeElement>
bool binaryTree<treeElement>::retrieve
    (const treeElement& el) const
{
     return retrieve(root, el);
} // retrieve
```

BinaryTree.cpp

```
// Retrieves for the item with same key as el
// in the subtree pointed to by aRoot. Called
// by public search.
// Pre : el and aRoot are defined.
// Returns true if el's key is located,
// otherwise, returns false.
template<class treeElement>
bool binaryTree<treeElement>::retrieve
     (treeNode* aRoot, treeElement& el) const
{
   return true;
}
```

13.8 Efficiency of a Binary Search Tree

- Searching for a target in a list is ***O(N)***
- Time is proportional to the size of the list
- Binary Tree more efficient
  - cutting in half process
- Possibly not have nodes matched evenly
- Efficiency is $O(\log_2 N)$

13.9 Common Programming Errors

- Use the ***** de-referencing operator
- Operator **->** member
- ****p*** refers to the entire node
- ***p->x*** refers to member ***x***
- ***new*** operator to allocate storage
- ***delete*** de-allocates storage
- Watch out for run-time errors with loops
- Don’t try to access a node returned to heap