Justin Ciocoi

Oct. 11, 2023

CSCI 373 Textbook Notes

Recursion

- *Repetition* is a key feature of high level languages, and we have seen that repetition can be achieved through for and while loops
- Another way to achieve repetition is through **recursion**, which occurs whenever a function calls itself within its own definition
- **The Factorial Function**
  - Let us define the factorial function
    - $n! = 1$, if $n = 0$
    - $n! = n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 3 \cdot 2 \cdot 1$ if $n \geq 1$
  - For example,
    - $5! = 5 * 4 * 3 * 2 * 1 = 120$
  - From this, we can see,
    - $5! = 5 * (4!)$
    - and then, since $4! = 4 * 3 * 2 * 1$
      - $5! = 5 * 4 * (3!)$
  - So this leads to the following recursive definition of the recursive factorial function
    - factorial(n) $= 1$, if $n = 0$
    - factorial(n) $= n \cdot factorial(n-1)$, if $n \geq 1$
  - As we can see, this function contains one or more **base cases**
    - In this case, the base case is 1 when $n = 0$

- There is no circularity in this definition because each time the function is invoked, its argument is smaller by one

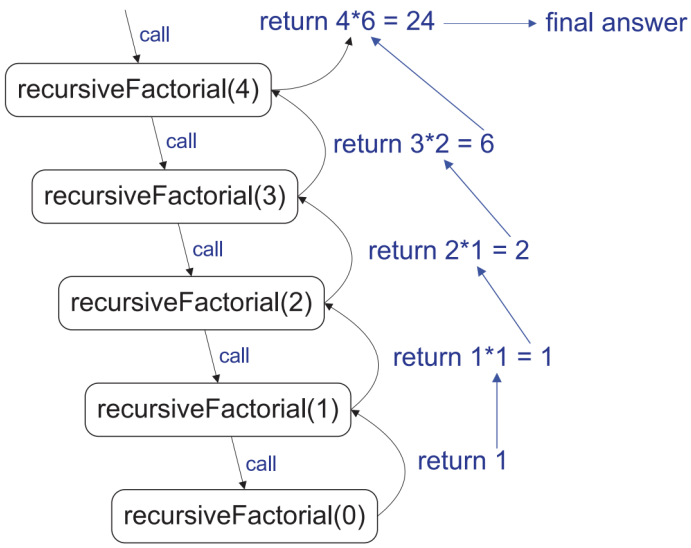
- **C++ Implementation of Recursion in the Factorial Function**
  - Note that in the following definition, no loops are necessary since recursion is used
  - ```
    int recursiveFactorial(int n)
    {
        if(n==0)
            return 1; //basis case which will always be called at the end
        else
            return n * recursiveFactorial(n-1); //recursive case
    }
    ```
  - This function can be illustrated using the following recursion trace
  - 

Figure 3.16: A recursion trace for the call recursiveFactorial(4).

- **Recursive Example using an English Ruler**
  - An English ruler is broken into intervals and each interval contains a set of *ticks*
  - These ticks are placed at intervals of $\frac{1}{2}$ inch, $\frac{1}{4}$ inch, and so on
  - As the size interval decreases by half, the tick length decreases by one
  - Below are some representations of English Rulers
  -
    ```

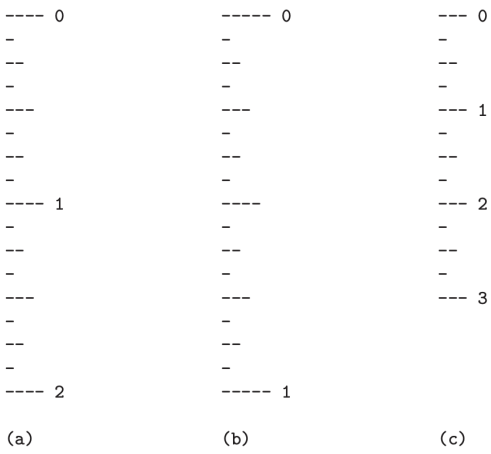


Figure 3.17: Three sample outputs of an English ruler drawing: (a) a 2-inch ruler with major tick length 4; (b) a 1-inch ruler with major tick length 5; (c) a 3-inch ruler with major tick length 3.

- The longest tick length of an English Ruler will be referred to as the *major tick length*
- One approach to drawing this consists of three functions
  - `drawRuler()` draws the entire ruler and takes the number of inches, `nInches`, and the major tick length, `majorLength` as arguments
  - The utility function, `drawOneTick()`, draws a single tick of the given length
  - `drawTicks`, which is the recursive function which draws the sequence of ticks within some interval
- Here is a C++ implementation of what is described above

```
void drawOneTick(int tickLength, int tickLabel = -1)
{
    for(int i=0; i<tickLength; i++)
        cout<<"-";
    if(tickLabel>=0)
        cout<<" "<<tickLabel;
    cout<<endl;
}
```

- This function draws one tick with an optional label

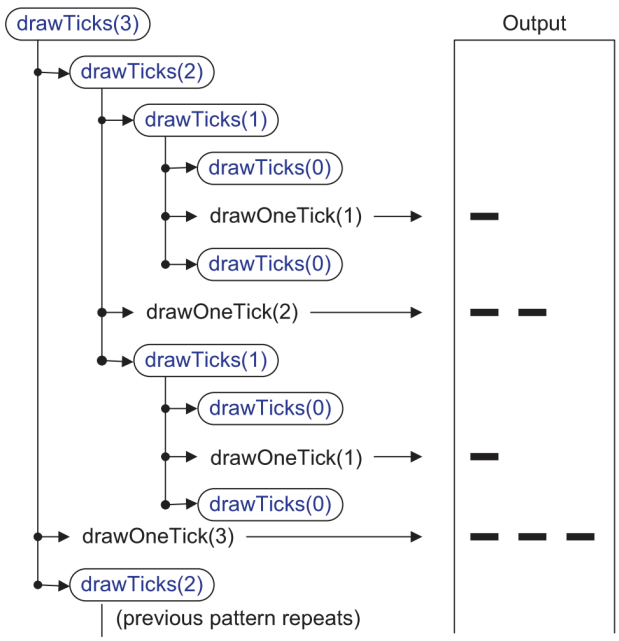
- ```
  void drawTicks(int tickLength)
  {
      if(tickLength>0)
      {
          drawTicks(tickLength-1);
          drawOneTick(tickLength);
          drawTicks(tickLength-1);
      }
  }
  ```
- This function recursively draws ticks between two major ticks
- ```
  void drawRuler(int nInches, int majorLength)
  {
      drawOneTick(majorLength, 0);
      for(int i=1; i<nInches; i++)
      {
          drawTicks(majorLength-1);
          drawOneTick(majorLength, i)
      }
  }
  ```
- This function can be used to draw the ruler as a whole
- 

Figure 3.18: A partial recursion trace for the call drawTicks(3). The second pattern of calls for drawTicks(2) is not shown, but it is identical to the first.

- The above recursion trace provides an illustration of what will occur when `drawRuler` is run with a major tick length of 3

- **More Examples of Recursion**
  - Recursion can be beneficial by allowing us to exploit a more *natural* form of repetition that does not involve complex nested loops or case analyses
  - *Example 3.1:* Modern OSes operate file-system directories in a recursive manner, meaning folders can be nested inside of folders in an arbitrarily deep fashion so long as there is sufficient space in memory
  - *Example 3.2:* The syntax in modern programming languages is most often defined in a recursive manner
- **3.5.1: Linear Recursion**
  - Linear recursion is the simplest form of recursion
  - Linear recursion refers to a recursive function that makes, at most, one recursive call each time that it is invoked
  - *Summing the elements of an array recursively*
    - Suppose we have an array, A, of n integers which we want to sum together
    - Since we know that the sum of all integers in A is equal to $A[0]$ when $n = 1$, we can solve this problem recursively with the following algorithm
      - ```
        Algorithm for LinearSum(A, n)

            Input: an integer Array, A, and int n >= 1
            Output: The sum of the first n integers in A

            if n=1 then
                return A[0]
            else
                return LinearSum(A, n-1)+A[n-1]
        ```
    - This illustrates one very important aspect of *all* recursive functions - the fact that it terminates
      - This can be fairly easily achieved by writing a non recursive statement for the base case, in this case the `if n=1` statement achieves this
  - In fact, an algorithm that employs linear recursion generally adheres to the following form
        ```

    - *Test for base cases*, where the function reaches a pre-defined base case for which a recursive call is not needed
        - Base cases should be defined such that every possible chain of recursive calls eventually reaches a base case
    - *Recursion*, where after testing for base cases, the function will recursively call itself
        - It might have to decide between different recursive steps, but a linear recursive algorithm should call itself recursively only once each time it is invoked
- Now, let us consider the recursion trace, or visual diagram representing a system's logic during a recursive linear summation
-

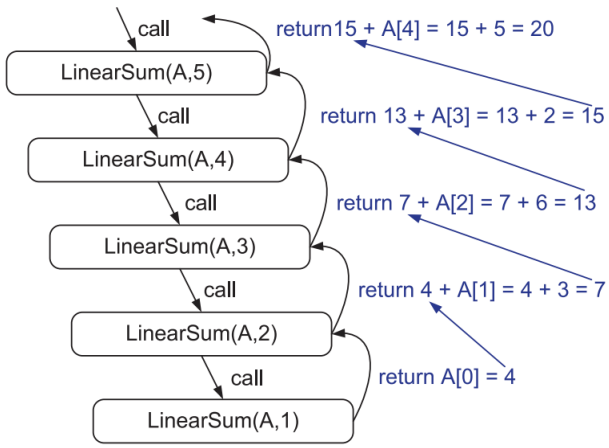


Figure 3.19: Recursion trace for an execution of LinearSum(A, n) with input parameters $A = \{4,3,6,2,5\}$ and $n = 5$.

- Recursive algorithms can, however, take up more space in the memory due to their need to store each prior recursive call until the function terminates
- Therefore, it can sometimes be useful to be able to derive non-recursive algorithms from recursive ones
- *Tail recursion* occurs when a recursive algorithms initiates the recursive call as the last thing it does other than base case evaluation
- Algorithms that utilize tail recursion are simple to convert from recursive to non-recursive
    - This can be achieved by iterating through the recursive calls rather than calling them explicitly
- Here is the algorithm for `IterativeReverseArray()`

```
Algorithm: IterativeReverseArray(A, i, j)

    Input: An array A, and non-negative integer indices i and j
    Output: Reversal of A from i to j

    while i<j
        Swap A[i] and A[j]
        i <- i+1
        j <- j-1
    return
```

- 3.5.2: Binary Recursion
  - When a function makes two recursive calls, we can refer to this as *binary recursion*
  - Let us look at the algorithm for a Binary sum
  - ```
    Algorithm BinarySum(A, i, n)

        Input: An array A an integers i and n
        Output: The sum of the n integers in A starting at i

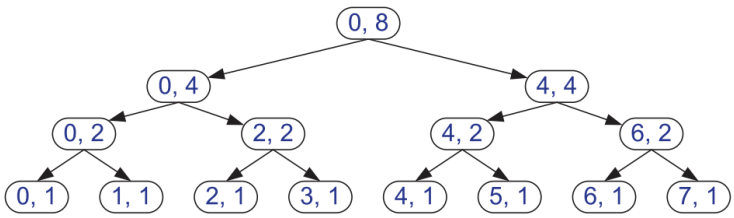
        if n=1
            return A[i]
        return
            BinarySum(A, i, [n/2]) + BinarySum(A, i+[n/2], [n/2])
    ```
  - Below is the trace for Binary Sum
    - 

      Figure 3.20: Recursion trace for the execution of BinarySum(0,8).

- **Computing Fibonacci Numbers via Binary Recursion**
    ```

- ```
  Algorithm BinaryFib(k):

      Input: Nonnegative integer k
      Output: The kth Fibonacci number Fk

      if k ≤ 1 then
          return k
      else
          return BinaryFib(k-1) + BinaryFib(k-2)
  ```
- However, this is of time complexity $O(n^2)$, because the number of recursive calls more than doubles with each consecutive index
- Therefore, it is actually more efficient to compute the k_{th} Fibonacci number using *linear recursion*

- **Computing Fibonacci Numbers via Linear Recursion**
  - ```
    Algorithm LinearFibonacci(k):

        Input: A nonnegative integer k
        Output: Pair of Fibonacci numbers (Fk,Fk-1)

        if k ≤ 1 then
            return (k,0)
        else
            (i, j)← LinearFibonacci(k-1)
            return (i+ j, i)
    ```
  - For this algorithm, the time complexity is $O(n)$ so it is far more efficient than binary recursion for Fibonacci calculations

- **3.5.3: Multiple Recursion**
  - If we generalize the jump from linear to binary recursion, we can arrive at *multiple recursion*
    - Multiple recursion algorithms may make multiple recursive calls, with that number being possibly more than two
  - Below is the algorithm and recursion trace for an algorithm written to solve *summation puzzles* where different letters represent integers in an equation

```
Algorithm PuzzleSolve(k,S,U):

    Input: An integer k, sequence S, and set U
    Output: An enumeration of all k-length extensions to S using elements
in U
            without repetitions

    for each e in U do
        Remove e from U {e is now being used} Add e to the end of S
        if k = 1 then
            Test whether S is a configuration that solves the puzzle
            if S solves the puzzle then
                return “Solution found: ” S
        else
            PuzzleSolve(k-1,S,U)
        Add e back to U {e is now unused}
        Remove e from the end of S
```

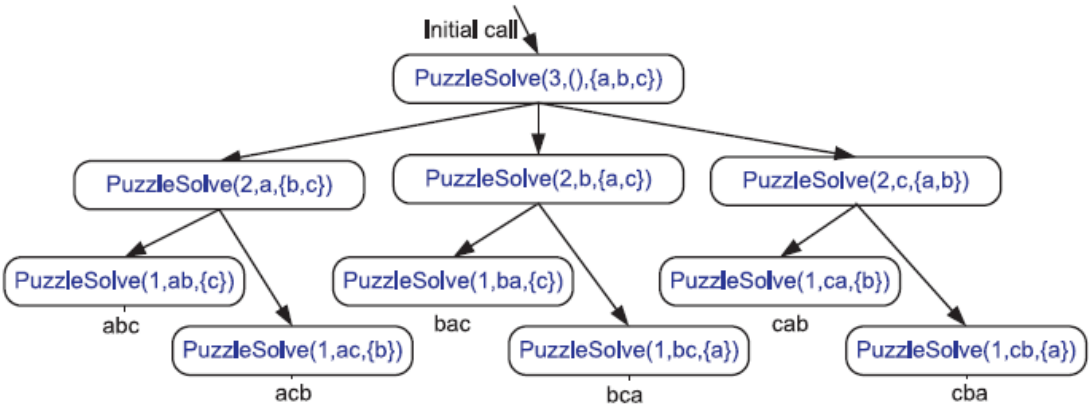


Figure 3.21: Recursion trace for an execution of PuzzleSolve$(3,S,U)$, where S is empty and $U = \{a,b,c\}$. This execution generates and tests all permutations of a, b, and c. We show the permutations generated directly below their respective boxes.

Justin Ciocoi

Oct. 5, 2023

CSCI 373 Class Notes

Advanced Data Structures and Algorithms

Singly Linked Lists

- Data is divided into nodes, which each contain *one* element
- Each node contains its own internal data as well as a link or pointer to the next node of the list
- Moving through the list using the sequential next pointers is called *pointer-hopping*
- Two nodes in a singly linked lists have special classifications
    - The *head* at the front of the list which is not pointed to by anything
    - The *tail* at the end of the list which points to a NULL value
- The structure is known as a singly linked list because each node stores a single link
- They can be maintained in an order and, unlike arrays in C++, do not have a predetermined size
- **Interface of StringNode Class**
    - ```
      class StringNode
      {
          private:
              string elem;
              StringNode* next;

              friend class StringLinkedList;
      };
      ```
- **Interface of StringLinkedList Class**
      ```

```
class StringLinkedList
{
    public:
        StringLinkedList();
        ~StringLinkedList();
        bool empty() const;
        const string& front() const;
        void addFront(const sstring& e);
        void removeFront();

    private:
        StringNode* head;
};
```

- **Insertion into a Singly Linked List**
  - You can insert at the head by:
    - Making a new node
    - Storing the address of the head in the next pointer of the new node
    - Setting the head variable to the new node
    - Below is the code representation of the above insertion algorithm
      - ```
        StringNode* v = new StringNode;

        v->next = head;

        head=v;
        ```

```
    ```

    ```
```

- You can insert at the end by:
  - Navigating the list to reach the last node
  - Create a new node
  - Attach the new node to the back of the last node
```

    - Ensure the new node points to NULL, which makes it the tail
    - Below is the code representation of the above insertion algorithm

        ```
        v=head;
        while(v->next != NULL)
            v=v->next;

        Node *n = new Node();

        v->next = n;

        n->next = NULL
        ```

- You can insert in the middle by:
    - Assuming we have a pointer to node `v`
    - We want to insert a node between `v` and `w` where `w` is a node immediately after `v`
    - First, create a new node, `z`
    - Copy the pointer to `w` to `z`
    - Make the pointer of `v` point to `z`
    - Below is the code representation of the above insertion algorithm

        ```
        Node *z = new Node();
        z->next = v->next;
        v->next = z;
        ```

Justin Ciocoi

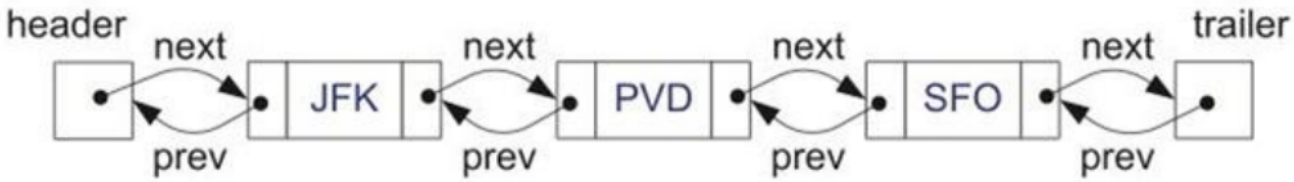
Oct. 5, 2023

CSCI 373 Class Notes

Advanced Data Structures and Algorithms

Doubly Linked Lists

- In a singly linked list, it takes long to remove any node other than the head
- To alleviate this issue, in a doubly linked list, pointers go in both directions
    - Each node stores two pointers, *next* and *prev*
- The *header* and *trailer* can be dummy nodes with pointers to the head and tail to provide quick access to these nodes
- The following is an example of a doubly linked list



Insertion in a Doubly Linked List

- Let us assume that we have a node, `v`, in a doubly linked list
    - `v` is not the trailer of the doubly linked list
- Now, we assume that `z` is a new node that we want to insert immediately after `v`
- Currently, we will assume `w` is the node immediately following `v`
- To insert a new element, we must first give it next and previous pointers to the appropriate elements and then re-define the next pointer for `v` and prev pointer for `w`
- Below is the code representation for this

```
z->prev=v;
z->next=w;
w->prev=z;
v->next=z;
```

- To remove, simply delete the node `z` and re-define the pointers between `v` and `w`

Interface of a Node in Doubly Linked List

```
typedef string Elem;
class dNode
{
    private:
        Elem elem;
        dNode* prev;
        dNode* next;
        friend class dLinkedList;
};
```

Interface of Doubly Linked List

```
class dLinkedList
{
    public:
        dLinkedList();
        ~dLinkedList();
        bool empty() const;
        const Elem& front() const;
        const Elem& back() const;
        void addFront(const Elem& e);
        void addBack(const Elem& e);
        void removeFront();
        void removeBack();
    private:
        dNode* header;
        dNode* trailer;
    protected:
        void add(dNode* v, const Elem& e);
        void remove(dNode* v);
}
```

Justin Ciocoi

Oct. 5, 2023

CSCI 373 Class Notes

Advanced Data Structures and Algorithms

Circularly Linked Lists

- It basically uses the same node structure as a singly linked list
- However, the list is circular, which means the last node of the list points back at the first node
- If a circularly linked list contains **only one** element, the node will point to *itself*
- Travel across a circularly linked list can only be done unidirectionally
- A cursor node is defined to remember the current node that we are in
- We also define a front and back node, although we don't need to implement them in code
    - Since its a circle, the program doesn't really care which is the *front* or *back* so we use this only for our understanding as programmers
- **Functions in a circularly linked list (API)**
    - `back()`
        - Return element referenced by the cursor
        - Empty list returns error
    - `front()`
        - Return element immediately after the cursor
        - Empty list returns error
    - `advance()`
        - Advance cursor to next node in list
    - `add(e)`

        - Insert a new node immediately after the cursor
        - If the list is empty, the node becomes the cursor and its next pointer points to itself
    - `remove()`
        - Remove the node immediately after the cursor
        - If cursor is the only node, then it is removed and the cursor is set to `NULL`
    - `add()` and `remove()` operate on the circularly linked list in a *Last-in First-out* stack data structure where code such as:
    - ```
      add(x);
      remove();
      add(y);
      remove();
      add(n);
      add(m);
      remove();
      remove();
      ```
    - will result in no change in the original circularly linked list
- **Interface of the node in a circularly linked list**
    - ```
      typedef string Elem;
      class cNode
      {
          Elem elem;
          cNode* next;

          friend class CircleList;
      };
      ```
- **Interface of the circularly linked list**

- ```
  class CircleList
  {
      public:
          CircleList();                    //sets cursor to null
          ~CircleList();                   //remove nodes one by one
          bool empty() const;
          const Elem& front() const;
          const Elem& back() const;
          void advance();
          void add(const Elem& e);
          void remove();

      private:
          cNode** cursor;
  };
  ```
  ```

Justin Ciocoi

Nov. 1, 2023

CSCI 373 Textbook Notes

Chapter 5: Stacks, Queues, and Dequeues

5.1: Stacks

- A *stack* is a container of objects that are inserted and removed according to the *last-in, first-out (LIFO)* principle
- The name, "stack," was derived from the metaphor of a stack of plates in a spring-loaded cafeteria plate dispenser
- **5.1.1: The Stack Abstract Data Type**
    - Stacks are fairly simple in the world of data structures, as they are one of the simplest, however they are also among the most important data structures due to their prevalence in application development and software engineering
    - Formally, a stack is an abstract data type that supports the following operations
        - `push(e)`: Insert element `e` at the top of the stack
        - `pop()`: Remove the top element form the stack
        - `top()`: Return a reference to the top element on the stack, without removing it
    - Additionally, we can define the following operations
        - `size()`: Return the number of elements in the stack
        - `empty()`: Return true if the stack is empty and false otherwise
- **5.1.2: The STL Stack**
    - The *Standard Template Library* provides programmers with an implementation of a stack
    - This implementation is based on the STL `vector` class

- In order to declare an object of type `stack` it is necessary to first include the definition file, which is called "stack"
- As with `string` or `vector` types, you must use `std::stack` or provide a `using` statement at the beginning of your code
- For example, the following declares a stack of integers

```
#include <stack>

using std::stack;

stack<int> myStack;
```

- The type of the elements within a stack is referred to as the stack's *base type*
- An STL stack dynamically resizes itself as new elements are pushed on
- The functions for an abstract stack type that we defined earlier also apply to the STL implementation, with one small difference
  - The functions `pop()` and `top()` do not throw an exception when applied to an empty stack and it is thus up to the programmer to be sure that such illegal memory access does not occur

- **5.1.3: A C++ Stack Interface**
  - Let us display an example of a C++ stack interface for a templated data type, `E`

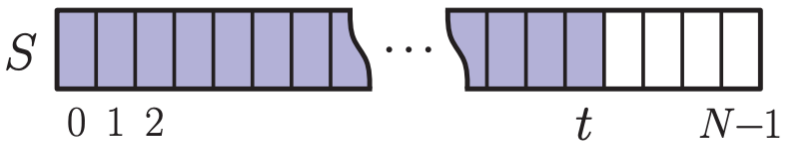
```
template <typename E>
class Stack
{
    public:
        int size() const;
        bool empty() const;
        const E& top() const throw(StackEmpty);
        void push(const E& e);
        void pop() throw(StackEmpty);
};
```

  - Here, we can note that the `size()`, `empty()`, and `top()` functions are all declared to be constant, which informs the compiler that they do not alter the contents of the stack

- We should also note that the `pop()` operation does not return the popped value, and if the user wants to know this value, it is necessary to perform a `top()` operation first
- We can also see the error conditions for `pop()` and `top()`, and we can define the class `StackEmpty` in the code fragment below

```
//Exception for stack
class StackEmpty : public RuntimeException
{
    public:
        StackEmpty(const string& err) : RuntimeException(err)
        {}
};
```

- **5.1.4: A Simple Array-Based stack Implementation**



- We can implement a stack easily by storing its elements in an array
- Specifically, this implementation would contain an N- element array, S, as well as an integer variable, t, that gives the index of the top element in array S
- To illustrate, let us first show the interface for the ArrayStack class

```
template <typename E>
class ArrayStack
{
        enum{DEF_CAPACITY = 100};
    public:
        ArrayStack(int cap = DEF_CAPACITY);
        int size() const;
        bool empty() const;
        const E& top() const throw(StackEmpty);
        void push(const E& e) throw(StackFull);
        void pop() throw(StackEmpty);

    private:
        E* S;
        int capacity;
        int t;
}
```

- Now, let us implement the member functions of the above class in C++

```
template <typename E> ArrayStack<E>::ArrayStack(int cap)
: S(new E[cap]), capacity(cap), t(-1) {}

template <typename E> int ArrayStack<E>::size() const
{
    return (t+1);
}

template <typename E> bool ArrayStack<E>::empty() const
{
    return (t<0);
}

template <typename E> const E& ArrayStack<E>
::top() const throw(StackEmpty)
{
    if(empty()) throw StackEmpty("Top of Empty Stack");
    return S[t];
}

template <typename E> void ArrayStack<E>
::push(const E& e) throw(StackFull)
{
    if(size()==capacity) throw StackFull("Push to Full Stack")
    S[++t] = e;
}

template <typename E> void ArrayStack<E>
::pop() throw(StackEmpty)
{
    if(empty()) throw StackEmpty("Pop from Empty Stack");
    --t;
}
```

- Before moving on, we should also consider the time complexity of the various stack class member functions)

Operation	*Time*
size	$O(1)$
empty	$O(1)$
top	$O(1)$
push	$O(1)$
pop	$O(1)$

- **5.1.5: Implementing a Stack with a Generic Linked List**
  - In this section, we will explore how we can implement the stack abstract data type in C++ using the generically linked list sLinkedList, which was defined earlier in the text
  - Instead of using a generic templated function, we will use a string type sLinkedList for ease of understanding and avoiding syntactic messiness
  - Here is the interface for this class
    - ```
      typedef string Elem;
      class LinkedStack
      {
          public:
              LinkedStack();
              int size() const;
              bool empty() const;
              const Elem& top() const throw(StackEmpty);
              void push(const Elem& e);
              void pop() throw(StackEmpty);
          private:
              sLinkedList<Elem> S;
              int n;
      };
      ```
  - Here, the principal data member is the generic linked list of type `<Elem>`, called `S`
    - Since this class does not provide a member function `size()`, we will store the current size in the variable `int n`
  - Now, let us define the implementation of the above interface
      ```

```
LinkedStack::LinkedStack()
: S(), n(0) {}

int LinkedStack::size() const
{
    return n;
}

bool LinkedStack::empty() const
{
    return n==0;
}

const Elem& LinkedStack::top() const throw(StackEmpty)
{
    if(empty()) throw StackEmpty("Top of Empty Stack")
    return S.front();
}

void LinkedStack::push(const Elem& e)
{
    ++n;
    S.addFront(e);
}

void LinkedStack::pop() throw(StackEmpty)
{
    if(empty()) throw StackEmpty("Pop from Empty Stack")
    --n;
    S.removeFront();
}
```

- **5.1.6: Reversing a Vector Using a Stack**
  - We can use the stack data type to reverse the elements in a vector, thereby producing a non-recursive algorithm for the array reversal problem
  - Here, the basic idea is to push all the elements of the vector in order into a stack and fill the vector back up by popping the elements off of the stack
  - Below is the implementation in C++ of this algorithm

```
template <typename E>
void reverse(vector<E>& V)
{
    //init stack
    ArrayStack<E> S(V.size());

    //push elements onto stack
    for(int i=0;i<V.size();i++)
        S.push(V[i]);

    //pop elements off of stack and back onto vector
    for(int i=0;i<V.size();i++
    {
        V[i] = S.top();
        S.pop();
    }
}
```

- **5.1.7: Matching Parentheses and HTML Tags**
  - Arithmetic expressions and other code in C++ will often contain various pairs of grouping symbols, including
    - (and)
    - [and]
    - { and }
  - When using these symbols, each opening symbol must correspond with an appropriate closing symbol
  - Here we can use a stack, and each time we encounter an opening symbol, we can push it onto the stack
  - Then, when we encounter a closing symbol, we can pop the top of the stack and compare it to ensure they are the same grouping symbols
  - Assuming that the `push()` and `pop()` operations are implemented to run in constant time, $O(1)$, this algorithm will run in linear time $O(n)$

5.2: Queues

- Another fundamental type of data structure is the *queue*, which is a close relative of the stack class
- A queue is a container of elements that are inserted and removed according to the *first-in first-out (FIFO)* principle
- Usually, we say that elements enter the queue at the *rear* and are removed from the *front*
- **5.2.1: The Queue Abstract Data Type**
    - Formally, the queue abstract data type defines a container that keeps elements in a sequence as defined above
    - The *queue* abstract data type supports the following operations
        - `enqueue(e)` : Insert element `e` at the end of the queue
        - `dequeue()` : Remove element at the front of the queue
            - Error occurs if the queue is empty
        - `front()` : Return, but do not remove, a reference to the front element in the queue
            - Error occurs if the queue is empty
        - `size()` : Return the number of elements in the queue
        - `empty()` : Return true if the queue is empty and false otherwise
    - We can observe a table showing a series of queue operations and their expected outputs

Operation	***Output***	$front \leftarrow Q \leftarrow rear$
enqueue(5)	–	(5)
enqueue(3)	–	(5,3)
front()	*5*	(5,3)
size()	*2*	(5,3)
dequeue()	–	(3)
enqueue(7)	–	(3,7)
dequeue()	–	(7)
front()	*7*	(7)
dequeue()	–	()
dequeue()	*"error"*	()
empty()	*true*	()

- **5.2.2: The STL Queue**
  - The Standard Template Library (STL), provides an implementation of a queue
  - The underlying implementation here, as with the STL stack class, is based on the underlying vector class in the standard template library
  - Similar to the stack data type, the below code fragment declared a queue of float type variables
    - ```
      #include <queue>
      using std::queue;

      queue<float> myQueue;
      ```
  - As with vectors and stacks, the STL queue dynamically resizes itself as new elements are added
  - The STL queue class supports roughly the same operators as thee abstract interface we defined above, with slight semantic and syntactic differences
    - `size()` : Return the number of elements in a queue
    - `empty()` : Return true f the queue is empty and false otherwise
    - `push(e)` : Enqueue e at the rear of the queue
    - `pop()` : Dequeue the element at the front of the queue
    - `front()` : Return a reference to the element at the queue's front
    - `back()` : Return a reference to the element at the queue's back
  - Unlike the abstract interface, the STL queue class provides access to both the front and the back of the queue
  - Like the STL stack class, no exceptions are thrown for full or empty lists and it is up to the programmer to ensure that such illegal memory access does not occur
- **5.2.3: A C++ Interface**
  - Here we can define a templated interface for the abstract queue data type
      ```

```
template <typename E>
class Queue
{
    public:
        int size() const;
        bool empty() const;
        const E& front() const throw(QueueEmpty);
        void enqueue(cinst E& e);
        void dequeue() thro(QueueEmpty);
};
```

- We can define the error conditions by defining the QueueEmpty class, which is a subclass of the RuntimeException class

```
class QueueEmpty : public RuntimeException
{
    public:
        QueueEmpty(const string& err) : RuntimeException(err)
        {}
};
```

- **5.2.4: A Simple Array-Based Implementation**
  - Let us consider an implementation of an array, Q with N elements, such that we can represent Q in a queue
  - The main issue with such an implementation is deciding how to keep track of the front and rear of the queue
  - One possibility to solve this issue is to adapt our implementation from the stack class such that $Q[0]$ is the front of the queue, and the queue will grow from there
    - This, however, is not efficient since it requires we move every element in the array when we perform a `dequeue()` operation
  - *Using an Array in a Circular Way*
    - Let us define three variables, f, r, and n
      - f is the index of the cell of Q storing the front of the queue
        - If the queue is not empty, then this is the index of the element that will be removed by `dequeue()`
      - r is the index of Q following the rear of the queue
        - If the queue is not full, this is where new elements will be inserted by `enqueue()`
      - n is the current number of elements in the queue
    - Initially, we set $n = 0$ and $f = r = 0$, which indicates an empty queue

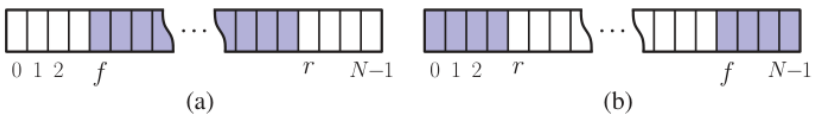
- When we dequeue an element, we decrement n and increment f to the next cell in Q
- When we enqueue an element, we increment r and increment n
- This approach will allow us to implement the dequeue and enqueue functions in constant time
- However, enqueuing and dequeuing a single element N, leads us to $f = r = N$, which gives us an out of bounds error for enqueue operations despite the availability of space in the queue
- Thus, we must implement the array in a circular fashion, as shown below
- 

Figure 5.4: Using array Q in a circular fashion: (a) the "normal" configuration with $f \leq r$; (b) the "wrapped around" configuration with $r < f$. The cells storing queue elements are shaded.

- *Using the Modulo Operator to Implement a Circular Array*
  - ```
    int size()
    {
        return n;
    }
    ```
  - ```
    bool empty()
    {
        return (n==0);
    }
    ```
  - ```
    Elem front()
    {
        if empty()
            throw QueueEmpty;
        return Q[f];
    }
    ```
    ```

```
void dequeue()
{
    if empty()
        throw QueueEmpty;
    f=(f+1)%N;
    n--;
}
```

```
void enqueue()
{
    if(size()==N)
        throw QueueFull;
    Q[r] = e;
    r = (r+1)%N;
    n++;
}
```

- 5.2.5: Implementing a Queue with a Circularly Linked List

```
//constructor
LinkedQueue::LinkedQueue()
: C(), n(0) {}

//number of items
int LinkedQueue::size() const
{
    return n;
}

//is the queue empty?
bool LinkedQueue::empty() const
{
    return n==0;
}

//return the front element
const Elem& LinkedQueue::front() const throw(QueueEmpty)
{
    if(empty())
        throw QueueEmpty("front of empty queue");
    return C.front();
}

//enqueue element at rear
void LinkedQueue::enqueue(const Elem& e)
{
    C.add(e);
    C.advance();
    n++;
}

//dequeue element at front
void LinkedQueue::dequeue() throw(QueueEmpty)
{
    if(empty())
        throw QueueEmpty("front of empty queue");
    C.remove();
    n--;
}
```

Justin Ciocoi

CSCI 373 Textbook Notes

Chapter 7: Trees

7.1: General Trees

- **7.1.1: Tree Definitions and Properties**
  - A *tree* is an abstract data type which stores elements hierarchically
  - Each element, with the exception of the top node has a parent element and zero or more children elements
  - The top node of a tree is generally called the *root*
  - Formally, a tree T can be defined as a set of nodes storing elements in a parent-child relationship with the following properties:
    - If T is non-empty, it has a special node, called its root, which has no parent
    - each node v of T different from the root has a unique parent node w; every node with parent w is a child of w
  - An *edge* of tree T is a pair of nodes (u, v) such that u is the parent of v or vice-versa
  - A *path* of T is a sequence of nodes such that any two consecutive nodes in the sequence form an edge
  - A tree is *ordered* if there is a linear ordering defined for the children of each node
- **7.1.2: Tree Functions**
  - The tree ADT stores elements at the nodes of the tree
  - Since nodes are internal aspects of our implementations, we don't allow direct access to them, opting instead to associate each node with a position object which provides public access to nodes
  - It is useful to overload the dereference operator, *, in order to return the element of node p when `∗p` is called

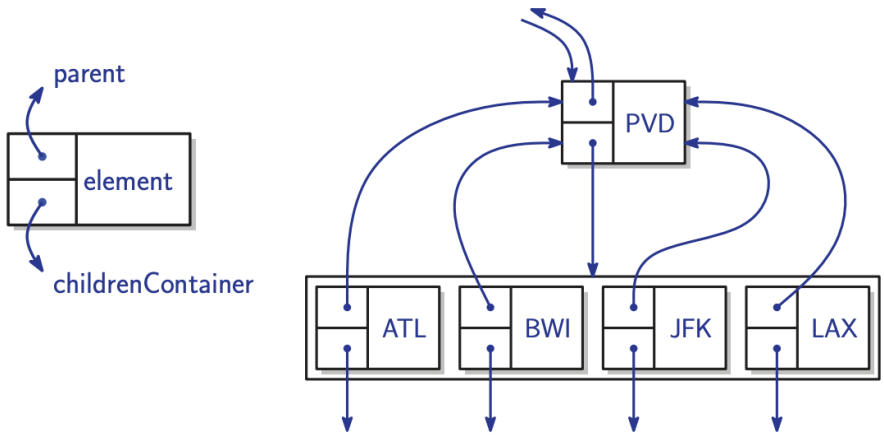
        - Given a position p of tree T, we can define the following:
            - `p.parent()` : Return the parent of p; error occurs if p is the root
            - `p.children()` : Return a position list containing the children of node p
            - `p.isRoot()` : Return true if p is the root and false otherwise
            - `p.isExternal()` : Return true if p is external and false otherwise
    - If a tree T is ordered, then the list provided by `p.children()` provides access to the children of node p in order
    - If p is external, then `p.children()` returns an empty list
    - For the tree ADT, we can also define the following functions
        - `size()` : Returns the number of nodes in the tree
        - `empty()` : Returns true if the tree is empty and false otherwise
        - `root()` : Returns a position for the tree's root; error occurs if the tree is empty
        - `positions()` : Returns a position list of all the nodes of the tree
- **7.1.3: A C++ Tree Interface**
    - First, let us present an interface for a position class which will represent a position in a tree

```
template <typename E>
class Position<E>
{
 public:
        E& operator*();
        Position parent() const;
        PositionList children() const;
        bool isRoot() const;
        bool isExternal() const;
}
```

    - Next, let us look at the C++ interface for a tree

```
template <typename E>
class Tree<E>
{
    public:
        class Position;
        class PositionList;
    public:
        int size() const;
        bool empty() const;
        Position root() const;
        PositionList positions() const;
}
```

- **7.1.4: A Linked Structure for General Trees**
  - A natural way to realize a tree T is to use a linked structure where we represent each node by a reference to its parent, its element, and its children
  - The fundamental idea of this can be seen in the below diagram where arrows function as pointers between data values



  - Now we can summarize the time complexity of the associated functions outlined above

Operation	*Time*
isRoot, isExternal	$O(1)$
parent	$O(1)$
children(p)	$O(c_p)$
size, empty	$O(1)$
root	$O(1)$
positions	$O(n)$

7.2: Tree Traversal Algorithms

- **7.2.1: Depth and Height**
  - Let p be a node of tree T
  - The *depth* of p is the number of ancestors of p excluding p itself
    - If p is the root, then the depth of p is 0
    - Otherwise, the depth is one plus the depth of the parent of p
  - Thus, we can achieve this with the following recursive function:

```
int depth(T, p)
{
    if(p.isRoot())
    {
         return 0;
    }
    else
    {
        return 1+depth(T,p.parent())
    }
}
```

    - The running time of this algorithm is $O(d_p)$, where d_p denotes the depth of the node p in the tree T
  - We can use a similar definition with the height of a tree, and also define the height of a node p recursively
    - If p is external, then the height of p is 0
    - Otherwise, the height of p is one plus the maximum height of a child of p

```
height1(T)
{
    for(p in T.positions)
    {
        if (p.isExternal())
            h=max(h,depth(T,p))
    }
    return h;
}
```

  - Which has a C++ implementation as follows

```
int height1(const Tree& T)
{
    int h = 0;
    PositionList nodes = T.positions();
    for(Iterator q = nodes.begin(); q != nodes.end;++q)
    {
        if(q->isExternal())
            h = max(h, depth(T, *q))
    }
    return h;
}
```

- However, this is an inefficient method and there exists a different C++ implementation that improves efficiency

```
int height2(const Tree& T, const Position &p)
{
    if(p.isExternal())
        return 0;
    int h = 0;
    PositionList ch = p.children();
    for(Iterator q = ch.begin();q != ch.end(); ++q)
        h = max(h, height2(T, *q);
    return 1+ h;
}
```

- **7.2.2: Pre-Order Traversal**
  - There are various methods for traversing across a tree structure, one of which is the pre-order traversal
  - This means that the root is processed, then the root's left child, and finally the root's right child
  - This process is done recursively down the entire tree until every node in the tree has been processed by the traversal algorithm
  - We can use the following C++ implementation of a pre-order traversal:

```
void preorderPrint(const Tree &T, const Position &p)
{
    cout<< *p;
    PositionList ch = p.children()
    for(Iterator q = ch.begin(); q != ch.end(); ++q)
    {
        cout<<" ";
        preorderPrint(T, *q);
    }
}
```

- **7.2.3: Post-Order Traversal**
    - In a post-order traversal, the left subtree is processed, then the right subtree, and finally, the root

```
void postorderPrint(const Tree &T, const Position &p)
{
    PositionList ch = p.children()
    for(Iterator q = ch.begin(); q != ch.end(); ++q)
    {
        cout<<" ";
        preorderPrint(T, *q);
    }
    cout<< *p;
}
```

7.3: Binary Trees

- A *binary tree* is an ordered tree in which every node has at most two children
- A binary tree must adhere to the following properties
    1. Every node has at most two children
    2. Each child node is labeled as being either a left child or a right child
    3. A left child precedes a right child in the ordering of children of a node
- A binary tree is proper if each node has either zero or two children
- A binary tree can be also be defined in a recursive manner:
    - A node r called the root of T and storing an element

- A binary tree, called the left subtree of T
- A binary tree, called the right subtree of T

- **7.3.1: The Binary Tree ADT**
  - Each node of the binary tree stores an element and is associated with a position object, which provides public access to nodes
  - By overloading the dereferencing operator, an element associated with position p can be accessed using $*p$
  - A position object, p, supports the following operations

    `p.left()` : Returns left child of p; Error if p is external

    `p.right()` : Returns right child of p; Error if p is external

    `p,parent()` : Returns parent of p; Error if p is root

    `p.isRoot()` : Returns true if p is the root and false otherwise

    `p.isExternal()` : Returns true if p is external and false otherwise
  - The binary tree ADT supports the same operations as the general tree, namely:
    - `size()` : Returns the number of nodes in the tree
    - `empty()` : Returns true if the tree is empty and false otherwise
    - `root()` : Returns a position for the tree's root; error occurs if the tree is empty
    - `positions()` : Returns a position list of all the nodes of the tree

- **7.3.2: A C++ Binary Tree Interface**
  - First, we can define the interface for the position objects which will be used by the class

```
template <typename E>
class Position <E>
{
 public:
 E& operator*();
 Position left() const;
 Position right() const;
 Position parent() const;
 bool isRoot() const;
 bool isExternal() const;
};
```

- Next, we define the interface for the actual tree class itself

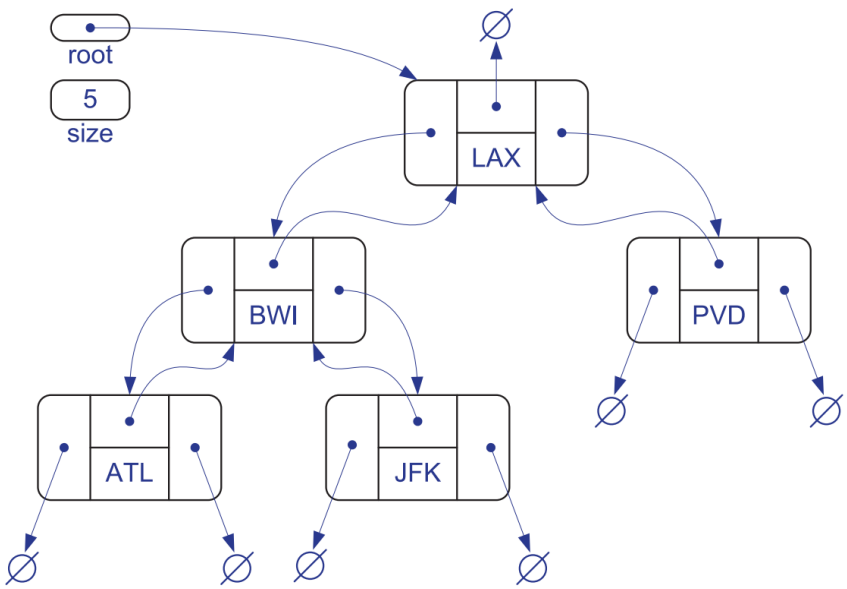
```
template <typename E>
class BinaryTree <E>
{
    public:
        class Position;
        class PositionList;
    public:
        int size() const;
        bool empty() const;
        Position root() const;
        PositionList positions() const;
};
```

- **7.3.3: Properties of Binary Trees**
  - Let T be a non-empty binary tree
  - Let n denote the number of nodes
  - Let n_E denote the number of external nodes
  - Let n_I denote the number of internal nodes
  - Let h denote the height of the tree
  - Then, a binary property will adhere to the following properties:
    1. $h + 1 \leq n \leq 2^{h+1} - 1$
    2. $1 \leq n_E \leq 2^h$

3. $h \leq n_I \leq 2^h - 1$
4. $log(n+1) - 1 \leq h \leq n - 1$

- If a tree is proper, then the following conditions will also be satisfied
  1. $2h + 1 \leq n \leq 2^{h+1} - 1$
  2. $h + 1 \leq n_E \leq 2^h$
  3. $h \leq n_I \leq 2^h - 1$
  4. $log(n+1) - 1 \leq h \leq (n-1)/2$
- We also have the relationship that in a non-empty proper binary tree, the number of external nodes is exactly one more than the number of internal nodes

- **7.3.4: A Linked Structure for Binary Trees**
  - This structure will define each node of a binary tree T as a part of a linked structure where each node has 4 components
    - First is the element of the node
    - Next is the parent pointer which points to the node's parent
    - Finally, two more pointers left and right point to the nodes left and right children respectively



  - Here is the implementation for the node class

```
struct Node
{
    Elem elt;
    Node* parent;
    Node* left;
    Node* right;
    Node() : elt(), parent(NULL), left(NULL), right(NULL)
};
```

- Next we can define the position class

```
class Position
{
    private:
        Node* v;
    public:
        Position(Node* _v = NULL) : v(_v) {}
        Elem& operator*()
        {
            return v->elt;
        }
        Position left() const
        {
            return Position(v->left);
        }
        Position right() const
        {
            return Position(v->right);
        }
        Position parent() const
        {
            return Position(v->parent);
        }
        bool isRoot() const
        {
            return v->parent == NULL
        }
        bool isExternal() const
        {
            return v->left == NULL && v->right == NULL
        };
    typedef std::list<Position> PositionList;
}
```

- Finally, we will look at the Linked Binary Tree class itself as well as the implementations of its member functions

```
typedef int Elem;
class LinkedBinaryTree
{
    protected:
        // node declaration
    public:
        // position declaration
    public:
        LinkedBinaryTree();
        int size() const;
        bool empty() const;
        Position root() const;
        PositionList positions() const;
        void addRoot();
        void expandExternal(const Position& p);
        Position removeAboveExternal(conts Position& p);
    protected:
        void preorder(Node *v, PositionList& pl) const;
    private:
        Node* _root;
        int n;
};
```

- Here n is the total number of nodes in the tree
- Next we can see the implementations of some of the above member functions

```
LinkedBinaryTree::LinkedBinaryTree()
: _root(NULL), n(0) {}

int LinkedBinaryTree::size() const
{
    return n;
}

bool LinkedBinaryTree::empty() const
{
    return size() == 0;
}

LinkedBinaryTree::Position LinkedBinaryTree::root() const
{
    return Position(_root);
}

void LinkedBinaryTree::addRoot()
{
    _root = new Node;
    n=1;
}
```

- Binary tree updating functions
  - `expandExternal(p)` will transform p from an external node to an internal node by creating two new external nodes as left and right children of p respectively; an error occurs if p is an internal node
  - `removeAboveExternal(p)` will remove the external node p along with its parent, q and then replace q with the sibling of p; an error occurs either if p is internal or if it is the root
- Now lets look at a C++ implementation of the `expandExternal()` function

```
void LinkedBinaryTree::expandExternal(const Possition& p)
{
    Node* v = p.v;
    v->left = new Node;
    v->left->parent = v;
    v->right = new Node;
    v->right->parent = v;
    n += 2;
}
```

- And now the `removeAboveExternal()` C++ implementation

```
LinkedBinaryTree::Position
LinkedBinaryTree::removeAboveExternal(const Position& p)
{
    Node* w = p.v;
    Node* v = w->parent;
    Node* sib = (w == v->left ? v->right : v->left);
    if(v == _root}
    {
        _root = sib;
        sib->parent = NULL;
    }
    else
    {
        Node* gpar = v->parent;
        if(v==gpar->left)
            gpar->left = sib;
        else
            gpar->right = sib;
        sib->par = gpar;
    }
    delete w;
    delete v;
    n -= 2;
    return Position(sib);
}
```

- Let us snow also look at the `positions()` function

```
LinkedBinaryTree::PositionList LinkedBinaryTree::positions() const
{
    PositionList pl;
    preorder(_root, pl);
}
```

- Which utilizes the following `preorder()` function for pre-order traversal of trees

```
void LinkedBinaryTree::preorder(Node* v, PositionList& pl) const
{
    pl.push_back(Position(v));
    if(v->left != NULL)
        preorder(v->left, pl);

    if(v->right != NULL)
        preorder(v->right, pl);
}
```

- And finally, we can look at the various time complexities for these various functions in the below table

Operation	*Time*
left, right, parent, isExternal, isRoot	$O(1)$
size, empty	$O(1)$
root	$O(1)$
expandExternal, removeAboveExternal	$O(1)$
positions	$O(n)$

- **7.3.5: A Vector-Based Structure for Binary Trees**
  - A fairly simple structure for representing a binary tree T is based on a way of numbering of the tree's nodes
  - For any node v of tree T, left $f(v)$ be the integer defined as follows
    - If v is the root of T, then $f(v) = 1$
    - If v is the left child of node u, then $f(v) = 2f(u)$
    - If v is the right child of node u, then $f(v) = 2f(u) + 1$
  - You must omit the 0^{th} index from the vector in order for the above rules to work

    - In this binary tree implementation, time complexities of the functions is the same as the linked-structure based binary tree

- **7.3.6: Traversals of a Binary Tree**
    - *Pre-order traversal of a binary tree*
        - The pre-order traversal processes the root, followed by recursive processing of the left subtree, and then the right subtree
    - *Post-order traversal of a binary tree*
        - The post-order traversal will first recursively process the left subtree, then the right subtree, and finishes by processing the root
    - *In-order traversal of a binary tree*
        - The in-order traversal will recursively process the left subtree, then the root, and will finish by processing the right subtree

Justin Ciocoi

Dec. 13, 2023

CSCI 373 Textbook Notes

Chapter 8: Heaps and Priority Queues

8.1: The Priority Queue Abstract Data Type

- A *priority queue* is an abstract data type for stoting a collection of prioritized elements that supports arbitrary element insertion but supports removal of elements in order of priority
- This ADT is fundamentally different from the position-based data structures
- The priority queue stores elements according to their priorities and has no external notion of value *position*
- **8.1.1: Keys, Priorities, and Total Order Relations**
    - Formally, we will define a *key* as an object which is assigned to each object in a collection as a specific attribute for that object and can be used to identify, rank, or weigh that element
    - Each element does not necessarily have a unique key, and an application may even *change* an element's key in order to achieve the desired program
    - A priority queue needs a comparison rule that never contradicts itself
    - For this, we must define a (total order relation) which is to say that the comparison rule is defined for every pair of keys and it must satisfy the following properties
        - *Reflexive Property*: $k \leq k$
        - *Antisymmetric Property*: if $k_1 \leq k_2$ and $k_2 \leq k_1$, then $k_1 = k_2$
        - *Transitive Property*: if $k_1 \leq k_2$ and $k_2 \leq k_3$, then $k_1 \leq k_3$
    - If these three rules are satisfied, then the comparison rule will never lead to a comparison contradiction
    - A priority queue is a container of elements, each associated with a key

- The fundamental functions of a priority queue, P, are as follows
  - `insert(e)` : Insert the element, e, (with implicit associated key values) into P
  - `min()` : return an element of P with the smallest key value
  - `removeMin()` : Remove the element `min()` from P

- **8.1.2: Comparators**
  - A comparator implemented within a priority queue will have be in the following form:
    - `isLess(a, b)` : where `true` will be returned if $a < b$ and `false` otherwise

- **8.1.3: The Priority Queue ADT**
  - As an ADT, a priority queue P supports the following functions:
    - `size()` : Return the number of elements in P
    - `empty()` : Return `true` if P is empty and `false` otherwise
    - `insert(e)` : Insert the element, e, (with implicit associated key values) into P
    - `min()` : return an element of P with the smallest key value
    - `removeMin()` : Remove the element `min()` from P

Operation	*Output*	*Priority Queue*
insert(5)	–	{5}
insert(9)	–	{5,9}
insert(2)	–	{2,5,9}
insert(7)	–	{2,5,7,9}
min()	[2]	{2,5,7,9}
removeMin()	–	{5,7,9}
size()	3	{5,7,9}
min()	[5]	{5,7,9}
removeMin()	–	{7,9}
removeMin()	–	{9}
removeMin()	–	{}
empty()	*true*	{}
removeMin()	*"error"*	{}

- **8.1.4: A C++ Priority Queue Interface**

```
template <typename E, typename C>
class PriorityQueue
{
    public:
        int size() const;
        bool isEmpty() const;
        void insert(const E& e);
        const E& min() const;
        void removeMin();
};
```

- **8.1.5: Sorting with a Priority Queue**
  - One very important application of a priority queue is sorting
  - To implement priority queue sorting with a list L of unordered elements and a priority queue P, the following phases will be implemented
    1. Put the elements of L into an initially empty priority queue, P through a series of n insert operations, one for each element
    2. Extract the elements from P in non-decreasing order by a series of n combinations of `min()` and `removeMin()` operations, putting them back into L in order

8.3: Heaps

- An efficient realization of a priority queue uses a data structure called a *heap*, which allows us to perform both insertions and removals in logarithmic time
- A heap will do this by generally abandoning the idea of storing elements and keys in a list and opting to instead store elements and keys in a binary tree
- **8.3.1: The Heap Data Structure**
  - *Complete Binary Tree*: A heap, T with height h is a complete binary tree if levels $\{0, 1, 2, \ldots, h-1\}$ have the maximum number of nodes and the nodes at level h fill this level from left to right
  - A heap, T, storing n entries has a height:

$$h = \lceil \log n \rceil$$

- **8.3.2: Complete Binary Trees and Their Representation**

- A complete binary tree, T supports all the functions of the binary tree ADT, plus the following two functions:
  - `add(e)` : Add to T and return a new external node v storing element e such that the resulting tree is a complete binary tree with last node v
  - `remove()` : Remove the last node of T and return its element
- For adding, there are essentially two cases to consider
  - If the bottom level of T is not full, then `add()` will insert a new node on the bottom level of T immediately after the rightmost node at this level
  - If the bottom level is full, then `add()` will insert a new node as the left child of the leftmost node of the bottom level of T
- *A vector based representation*
  - For a complete binary tree, T, stored in a vector A such that node v in T is the element of A with an index of $f(v)$ defined by the following rules:
    - If v is the root of T, then $f(v) = 1$
    - If v is the left child of node u, then $f(v) = 2f(u)$
    - If v is the right child of node u, then $f(v) = 2f(u) + 1$
- This can be achieved in C++ using the following interface

```
template <typename E>
class CompleteTree
{
public:
    class Position;
    int size() const;
    Position left(const Position& p);
    Position right(const Position& p);
    Position parent(const Position& p);
    bool hasLeft(const Position& p) const;
    bool hasRight(const Position& p) const;
    bool isRoot(const Position& p) const;
    Position root();
    Position last();
    void addLast(const E& e);
    void removeLast();
    void swap(const Position& p, const Position& q);

    typedef typename std::vector<E>::iterator Position;

private:
    std::vector<E> V;

protected:
    Position pos(int i)
    {
        return V.begin()+i;
    }

    int idx(const Position& p) const
    {
        return p-V.begin();
    }
}
```