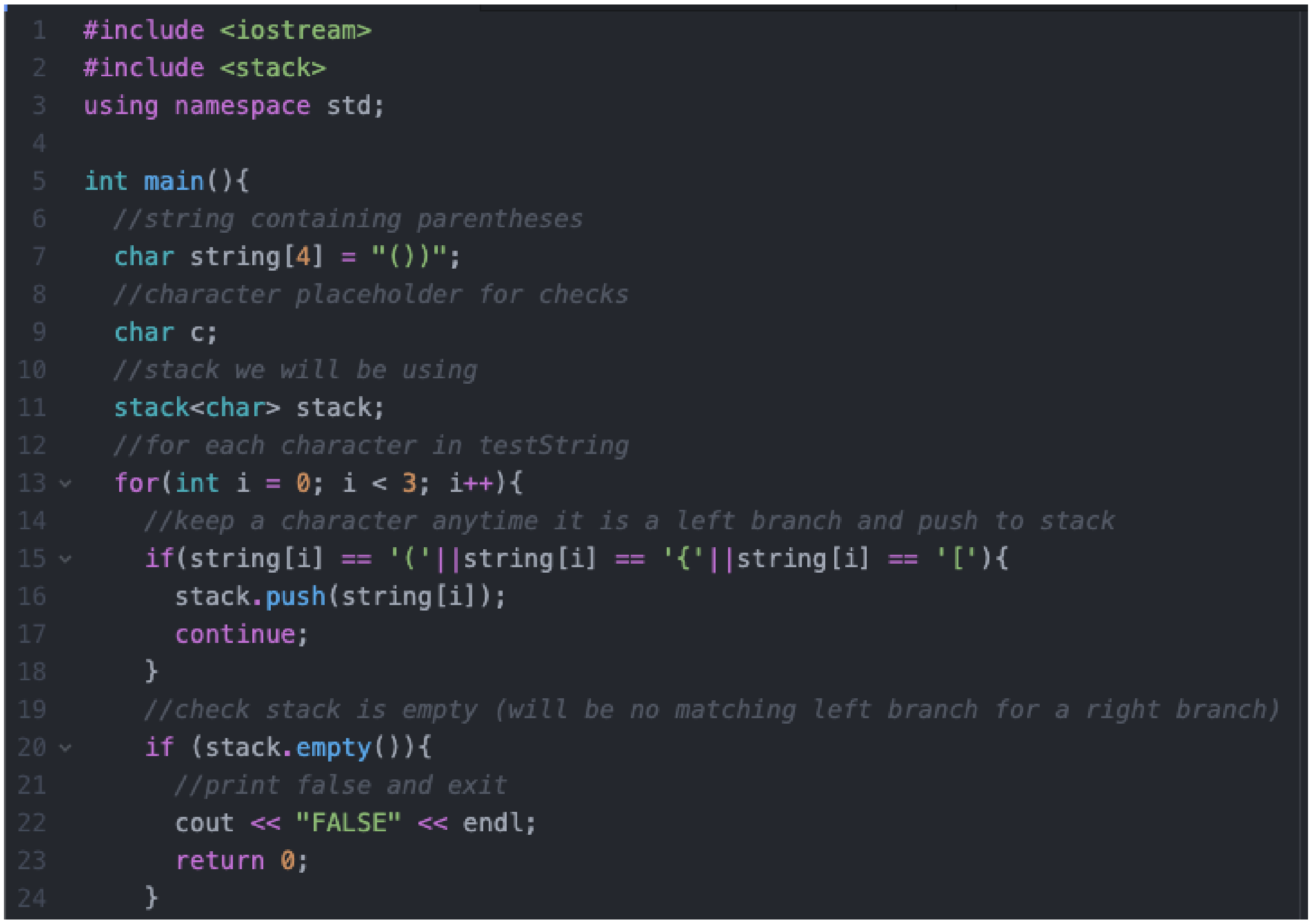
Ashlyn Cooper
CPSC 3120 Spring 2022
Test 1

Question 1:

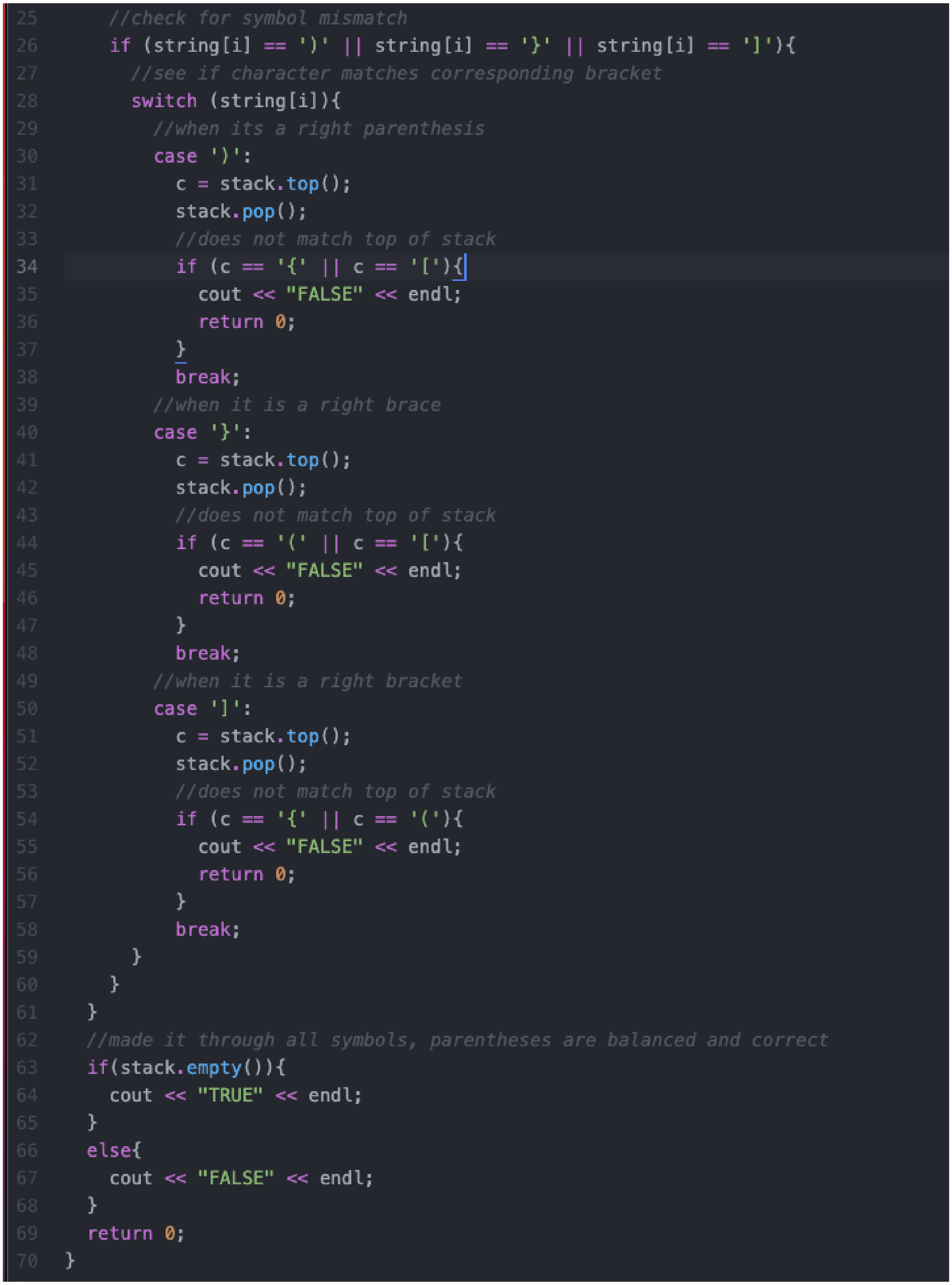
Pseudocode:

```
For each character in string:
     If character is an open symbol i.e. (,{,[, push to the stack

     if the stack is empty, print FALSE as no open symbol exists and
       exit

     if the closed symbol does not match the symbol at the top of the
       stack pop symbol at top from the stack, print FALSE and exit

After checking each character:
     If stack is empty print TRUE
     Otherwise print FALSE
```

Real Code in C++:



```cpp
#include <iostream>
#include <stack>
using namespace std;

int main(){
  //string containing parentheses
  char string[4] = "())";
  //character placeholder for checks
  char c;
  //stack we will be using
  stack<char> stack;
  //for each character in testString
  for(int i = 0; i < 3; i++){
    //keep a character anytime it is a left branch and push to stack
    if(string[i] == '('||string[i] == '{'||string[i] == '['){
      stack.push(string[i]);
      continue;
    }
    //check stack is empty (will be no matching left branch for a right branch)
    if (stack.empty()){
      //print false and exit
      cout << "FALSE" << endl;
      return 0;
    }
```



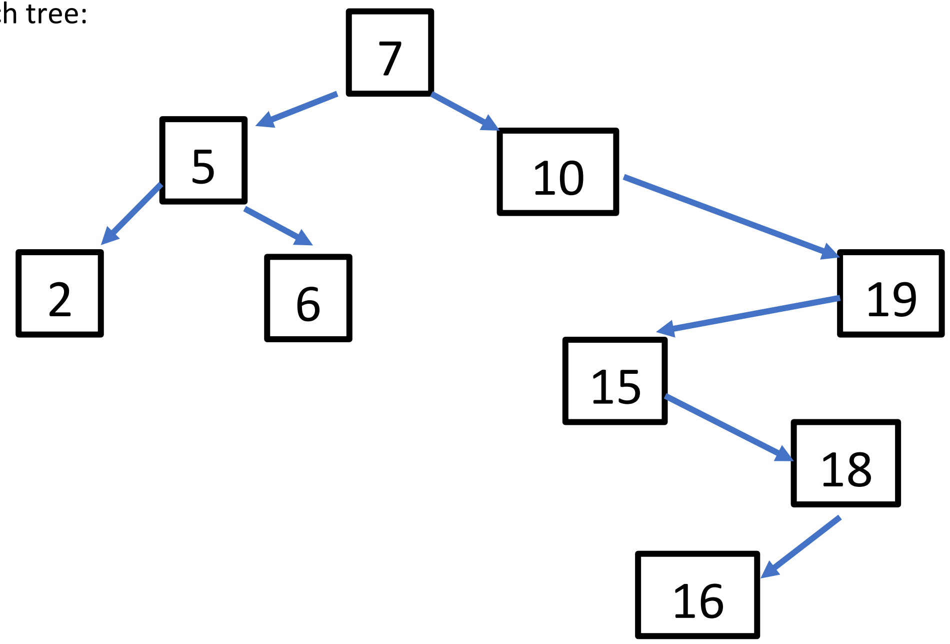
```cpp
    //check for symbol mismatch
    if (string[i] == ')' || string[i] == '}' || string[i] == ']'){
      //see if character matches corresponding bracket
      switch (string[i]){
        //when its a right parenthesis
        case ')':
          c = stack.top();
          stack.pop();
          //does not match top of stack
          if (c == '{' || c == '['){
            cout << "FALSE" << endl;
            return 0;
          }
          break;
        //when it is a right brace
        case '}':
          c = stack.top();
          stack.pop();
          //does not match top of stack
          if (c == '(' || c == '['){
            cout << "FALSE" << endl;
            return 0;
          }
          break;
        //when it is a right bracket
        case ']':
          c = stack.top();
          stack.pop();
          //does not match top of stack
          if (c == '{' || c == '('){
            cout << "FALSE" << endl;
            return 0;
          }
          break;
      }
    }
  }
  //made it through all symbols, parentheses are balanced and correct
  if(stack.empty()){
    cout << "TRUE" << endl;
  }
  else{
    cout << "FALSE" << endl;
  }
  return 0;
}
```

Question 2:

a) True
b) True
c) False
d) true
e) False

Question 3

Binary Search tree:



- Preorder Traversal: 7, 5, 2, 6, 10, 19, 15, 18, 16
- Inorder Traversal: 2, 5, 6, 7, 10, 15, 16, 18, 19
- Postorder Traversal: 2, 6, 5, 16, 18, 15, 19, 10, 7

Question 4:

a)
Pseudocode:

```
If int at position 0 > int at position 1, int at position 0 is
     max, int at position 1 is min
Else int at position 0 is min and int at position 1 is max

For each integer in array (starting at position 2):
     If int > max, set max to new int
     Else if int < min set min to new int
     else continue
```

Complexity of algorithm: $T(n) = 2(n-1) + 1$
Total Number of Element Comparisons:
- Best Case: n - 1
- Worst Case: 2n – 1

Array Characterization:
- Best Case: array is in ascending order with no equal elements i.e. 1, 2, 3, 4, 5, 6, etc
- Worst Case: The array is made up of integers whose 3rd – n elements are in between the first two elements. This would mean that both comparisons in the loop would need to be made for each element after the first two. i.e 1, 100, 99, 98, 97, 96 20, etc

b) Divide and conquer
Pseudocode:

```
Let i denote first index of array and j denote number of
     elements and mid denotes the midpoint index

If there is one element in the array (i = j), max and min is
     that element (i).

Else if there are two elements (i = j-1):
       o If 1st element is > 2nd, 1st is the max and 2nd is the
         min
       o Else, 2nd is the max, 1st is the min

Else:
       o we divide our array in half and get mid = i+j / 2
       o we call our function recursively on our array from i
         to mid
       o set max1 = current max and min1 = current min
       o call our function recursively on 2nd half of array
         (mid+1 to j)
       o if max < max1, set new max = max1
```

```
    o  if min > min1, set new min = min1
```

- Recurrence Relation: T(n) = 2T(n/2) + 2
- Solution to recurrence relation: T(n) = 3(n/2) - 2
- Compared Performance: The performance for the divide and conquer approach is better than the previous simpler approach. The number of comparisons is smaller. We also reduce the search region by half in each iteration.

Question 5:

Expression	**Value**	**Explanation**
sp[1].h	2	The pointer sp pointer to the second element of s (index 1). So sp[0] would be the 2nd element of s, making sp[1] the third element of s. In the third element of s, h = 2.
ip[5]	2	In this case we are looking for the 6th element (index 5) of ip. Ip is an array of integers that begins at the len element in s[0], which is the overall 3 integer in s. From that element we count off the next 5 integers in s to arrive at 2 or s[2].h
ip[8]-3	8	To find this value we go to the 9th element of ip or index 8. Ip[0] is the value at the address of s[0].len. From there we count up 8 integer elements in s to get to the 9th element. The 9th element is s[3].h or 11. From this value we subtract 3 to get 8.
&s	208	This expression returns the address where s is stored. It was given that array s begins at location 208 so that is what this returns.
ip[6]+6	9	To find this value we go to the 7th element of ip or index 6. Ip[0] is the value at the address of s[0].len. From there we count up 6 integer elements in s to get to the 7th element. The 7h element is s[2].len or 3. From this value we add 6 to get 9.
(sp+1)->w	1	First we go to the start of sp which is at s[1]. From there we jump forward 1 solid element moving to sp[1], which is technically s[2]. In that we get the value held in the w variable in s[2] which is 1.
*(ip+5)*10	20	We go to the location where ip starts, which is s[0].len. We then go forward 5 integers in s or to s[2].h. From there we dereference that address to get the value store or 2. Then we multiply by 10 to get 20.
(*(sp+2)).len-18	-6	We start at the address of sp, which is s[1]. Then we jump forward 2 solids to get to what is s[3] or sp[2]. We then get the value held in the length variable of s[3] or 12. Then we subtract 18 to get -6.

sp->w	5	We go to the location of sp, which is s[1]. Then we access the value in the w variable of s[1], which is 5.
ip	216	This returns the address of the start of ip. We know that ip begins at the address of s[0].len so we can start at the beginning of s or s[0] (which we know to be 208) and work our way to length, adding 4B to that value to get 216.

Question 6

Description:

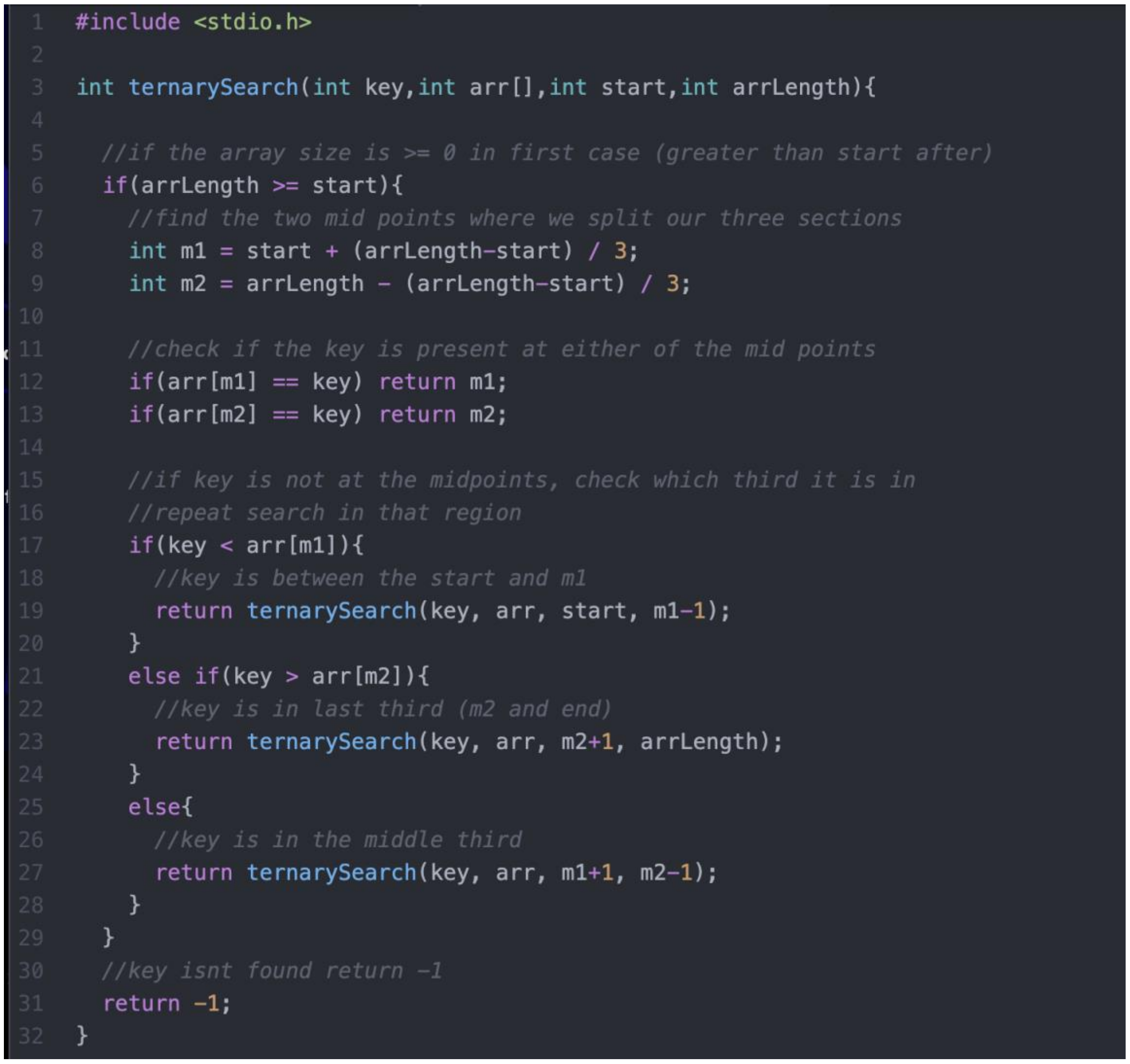
With Dijkstra's algorithm we assume that there is a path from the source vertex to every other vertex in the graph. We let the distance of the start vertex from the start vertex = 0. Then we let the distance of all other vertices from start be equal to infinity. Then we iteratively repeat through the following steps: 1) Visit the unvisited vertex with the smallest known distance from the source. 2) examine the unvisited neighbors of the current vertex. 3) For the current vertex calculate the distance of each neighbor from the source. 4) If the calculated distance is less than the known distance, we update the shortest distance and update the previous vertex for each updated distance. 5) We add the current vertex to a list of visited vertices. We do this repetition until all vertices are visited and we have a shortest path.

Shortest Path a to i: a -> b -> h -> i

- Step 1: visit a, explore neighbors b and c. S now contains a.
- Step 2: visit b, explore neighbors e, h, and i. S now contains a and b
- Step 3: visit h, explore neighbors i. S now contains a, b, h.
- Step 4: visit c, explore neighbors d and e. S now contains a, b, h, c
- Step 5: visit e, explore neighbors f and g. S now contains a, b, h, c, e
- Step 6: visit g, no unvisited neighbors to explore. S now contains a, b, h, c, e, g
- Step 7: visit d, no unvisited neighbors to explore. S now contains a, b, h, c, e, g
- Step 8: visit I, no unvisited neighbors to explore. S now contains a, b, h, c, e, g, i

Question 7:

1) Function in C:

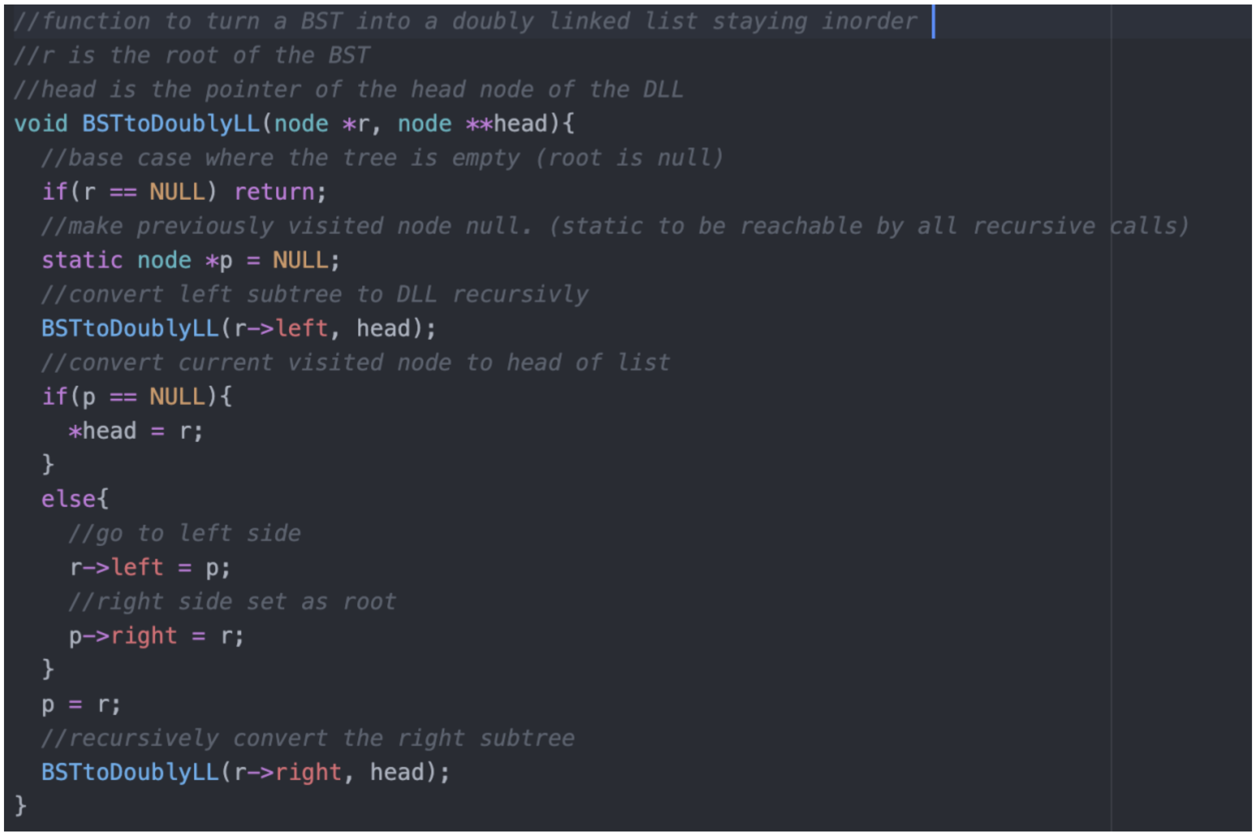


```c
#include <stdio.h>

int ternarySearch(int key,int arr[],int start,int arrLength){

  //if the array size is >= 0 in first case (greater than start after)
  if(arrLength >= start){
    //find the two mid points where we split our three sections
    int m1 = start + (arrLength-start) / 3;
    int m2 = arrLength - (arrLength-start) / 3;

    //check if the key is present at either of the mid points
    if(arr[m1] == key) return m1;
    if(arr[m2] == key) return m2;

    //if key is not at the midpoints, check which third it is in
    //repeat search in that region
    if(key < arr[m1]){
      //key is between the start and m1
      return ternarySearch(key, arr, start, m1-1);
    }
    else if(key > arr[m2]){
      //key is in last third (m2 and end)
      return ternarySearch(key, arr, m2+1, arrLength);
    }
    else{
      //key is in the middle third
      return ternarySearch(key, arr, m1+1, m2-1);
    }
  }
  //key isnt found return -1
  return -1;
}
```

2) Asymptotic behavior: θ(logn) . This is because we leave out the constants when writing asymptotic behavior in big O notation, so it is the same as binary search, even though it technically runs in $\log_3$n time. Ternary search does $\log_3$n recursive decisions. Even though the asymptotic complexity is the same as binary search, we should still use binary search over ternary search. This is because even though there is a smaller number of iterations, we must do more comparisons at each iteration, so ternary search is not more efficient.

3) Number of element comparisons ternary search: T(n) = $4c\log_3 n + c$
Number of element comparisons binary search: T(n) = $2c\log_2 n + c$
There is more insight into the comparison made in the previous section. From this we can see that the ternary search will have to do more comparisons because, in its worst case it will do more comparisons at each split and we are splitting into more sections as well.

Question 8



```
//function to turn a BST into a doubly linked list staying inorder
//r is the root of the BST
//head is the pointer of the head node of the DLL
void BSTtoDoublyLL(node *r, node **head){
  //base case where the tree is empty (root is null)
  if(r == NULL) return;
  //make previously visited node null. (static to be reachable by all recursive calls)
  static node *p = NULL;
  //convert left subtree to DLL recursivly
  BSTtoDoublyLL(r->left, head);
  //convert current visited node to head of list
  if(p == NULL){
    *head = r;
  }
  else{
    //go to left side
    r->left = p;
    //right side set as root
    p->right = r;
  }
  p = r;
  //recursively convert the right subtree
  BSTtoDoublyLL(r->right, head);
}
```

Bonus Questions

a) There are 144 ways 6 people can sit around a round table If 3 people want to sit next to each other. To solve this, we must figure out how our restricted participants must sit. The first restricted person has 6 seats to choose from. The next restricted person must sit next to the first, so they must sit on either side of them which leaves them with 2 options. The third restricted friend than has 2 ways to sit next to the friends, 1 on either end. Then the first unrestricted guest has 3 options to choose from once the first seats are taken. The next has 2 options and the last has 1. We multiply all of these options together which is 6 * 2 * 2 * 3 * 2 * 1 to get 144 ways.

b) There are 160 unique ways the committee can be formed so that no 2 Senators are form the same state. To start, we have 12 options with who we can choose. After that we have 10 options to choose from (because we eliminate the person we chose and the other Senator from their state). Then we finally have 8 to choose from for the third member following the same logic as the second member. We multiply these three numbers together to get 960 combinations, but these are not unique combinations and include some combinations where the state makeup is the same, we just chose a different senator. So, we divide 960 by 6 to eliminate the duplicates to get 160.