Data Structure
Tutorial 04

Prepared by Dr. Patrick Chan

1. Write a function that takes as input a general tree and returns the number of nodes in that tree. Write your function to use the `GenTree` and `GTNode` ADTs of Figure 6.2 (pp. 194).

Answer:

```
int gencount(GTNode<Elem>* subroot) {
  if (subroot == NULL) return 0
  int count = 1;
  GTNode<Elem>* temp = subroot->leftmost_child();
  while (temp != NULL) {
    count += gencount(temp);
    temp = temp->right_sibling();
  }
  return count;
}
```

or

```
int gencount(GTNode<Elem>* subroot) {
  if (subroot == NULL) return 0
return 1 +
       gencount(subroot->leftmost_child())+
       gencount(subroot->right_sibling());
}
```

2. Write an algorithm to determine if two general trees are identical.

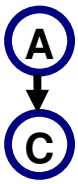
Answer:

```
// Return TRUE iff t1 and t2 are roots of identical
// general trees
template <class Elem>
bool Compare(GTNode<Elem>* t1, GTNode<Elem>* t2) {
  GTNode<Elem> *c1, *c2;
  if (((t1 == NULL) && (t2 != NULL)) ||
      ((t2 == NULL) && (t1 != NULL)))
    return false;
  if ((t1 == NULL) && (t2 == NULL)) return true;
  if (t1->val() != t2->val()) return false;
  c1 = t1->leftmost_child();
  c2 = t2->leftmost_child();
  while(!((c1 == NULL) && (c2 == NULL))) {
    if (!Compare(c1, c2)) return false;
    if (c1 != NULL) c1 = c1->right_sibling();
    if (c2 != NULL) c2 = c2->right_sibling();
}}
```

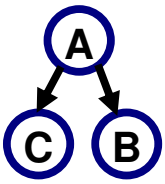
3. Using the weighted union rule and path compression, show the array for the parent pointer implementation that results from the following series of equivalences on a set of objects indexed by the values 0 through 15. Initially, each element in the set should be in a separate equivalence class. When two trees are the same size, make the root with greater index value be the childe of the root with lesser index value.

1. (A, C)	6. (J, L)	11. (I, K)
2. (B, C)	7. (M, O)	12. (I, H)
3. (D, E)	8. (D, J)	13. (H, A)
4. (D, B)	9. (E, O)	14. (K, P)
5. (D, F)	10. (G, H)	15. (K, N)

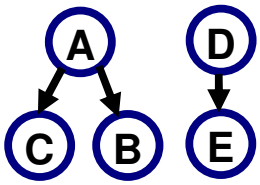
Answer:



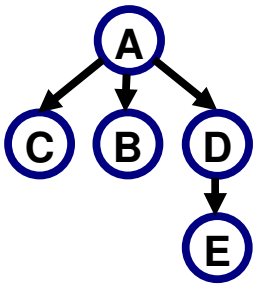
(A, C)



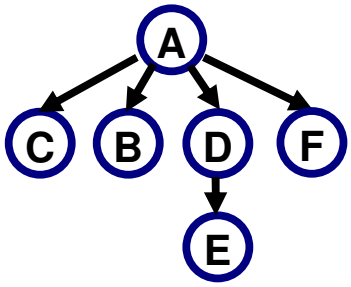
(B, C)



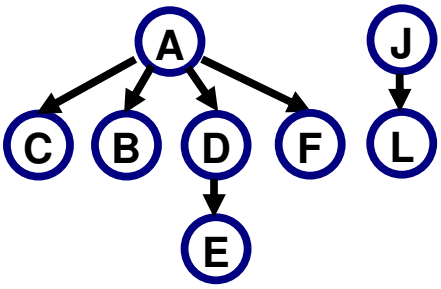
(D, E)



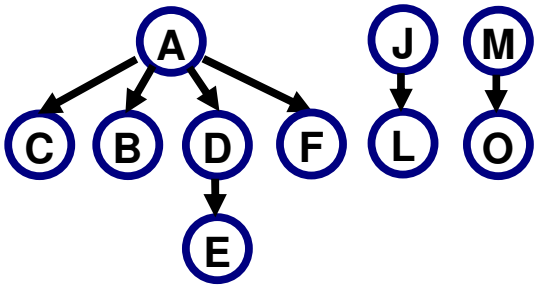
(D, B)



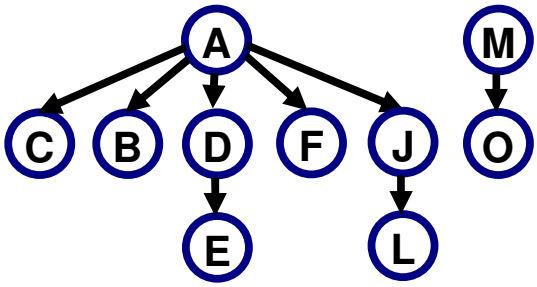
(D, F)



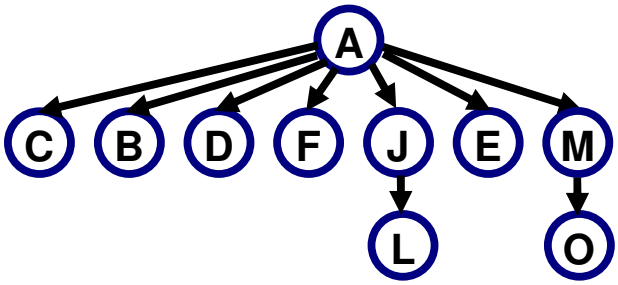
(J, L)



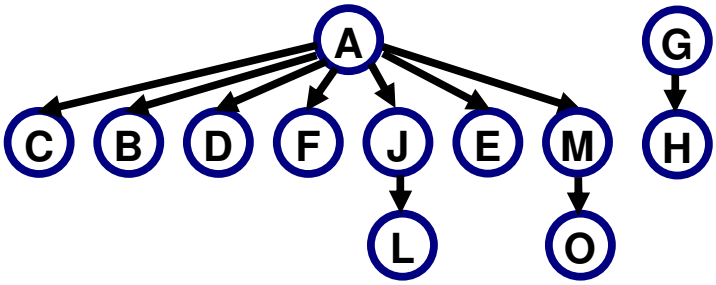
(M, O)



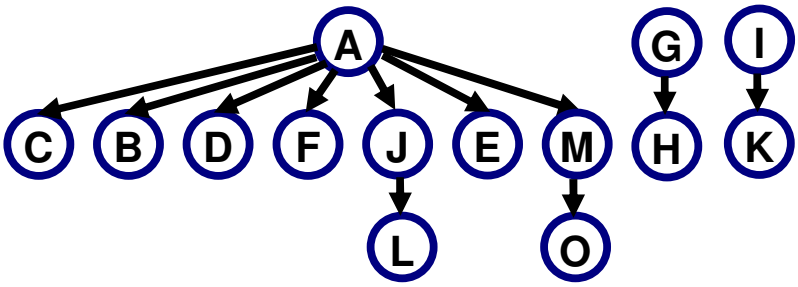
(D, J)


A
C
B
D
F
J
E
M
L
O

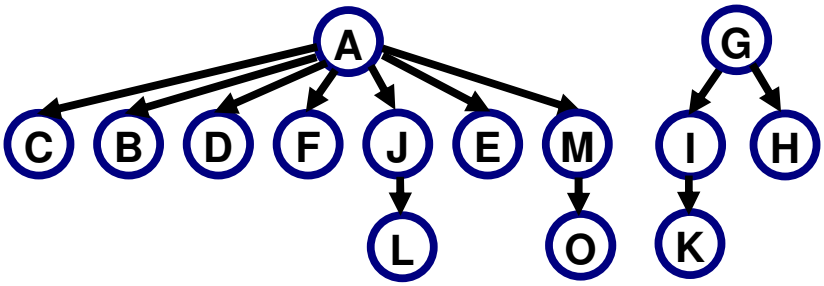
(E, O)


A
G
C
B
D
F
J
E
M
H
L
O

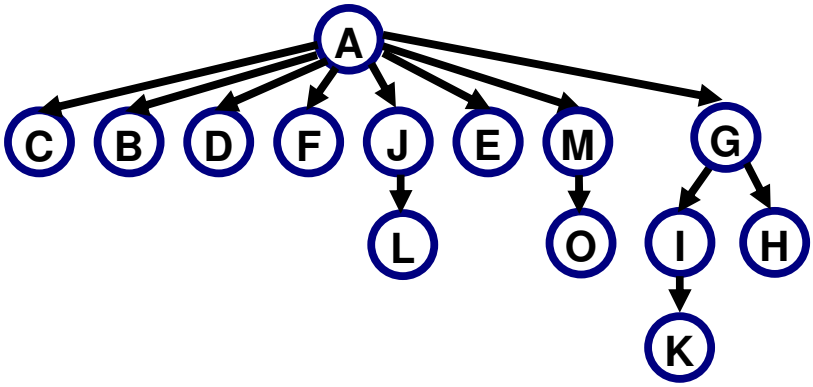
(G, H)


A
G
I
C
B
D
F
J
E
M
H
K
L
O

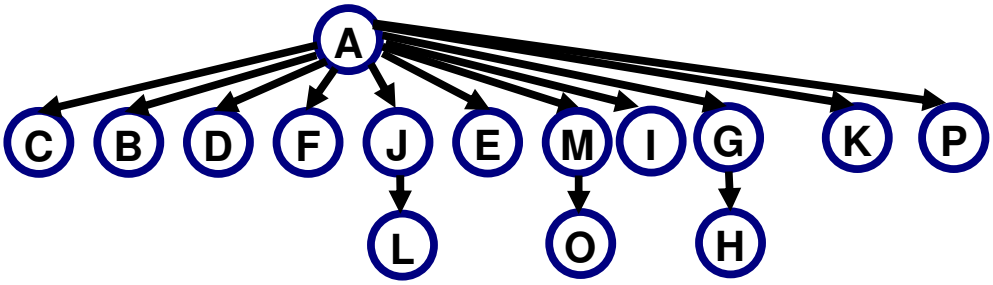
(I, K)


A
G
C
B
D
F
J
E
M
I
H
L
O
K

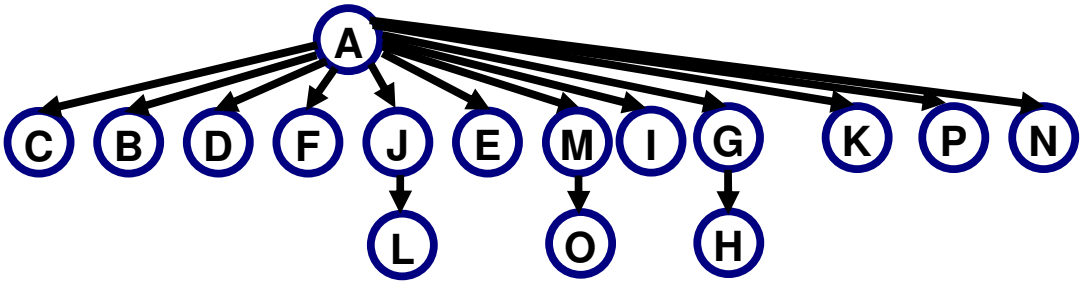
(I, H)



(H, A)



(K, P)



(K, N)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
/	0	0	0	0	0	0	6	0	0	0	9	0	0	12	0
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P

4. Analyze the fraction of overhead required by the following methods:
   i) List of Children Implementation
   ii) Left-Child/Right-Sibling Implementation
   iii) Dynamic Left-Child/Right-Sibling Implementation
   iv) Dynamic Node Implementation (array based)
   v) Dynamic Node Implementation (pointer)

Answer :

Let D as the size of the data value
P as the size of a pointer
I as the size of an index

i) Every node stores a data value, an index and a pointer to its list of children. Further, every child (every node except the root) has a record associated with it containing an index and a pointer. The overhead fraction is

$$(2P + 2I) / (D + 2P + 2I)$$

ii) For Left-Child/Right-Sibling Implementation, every node stores two pointers and a data value, for an overhead fraction of

$$(3I) / (D + 3I)$$

iii) For Dynamic Left-Child/Right-Sibling Implementation, every node stores two pointers and a data value, for an overhead fraction of

$$(2P) / (D + 2P)$$

iv) The first linked representation of Section 6.3.3 stores with each node a data value and a size field (denoted by S). Each child (every node except the root) also has a pointer pointing to it. The overhead fraction is thus

$$(S + P) / (D + S + P)$$

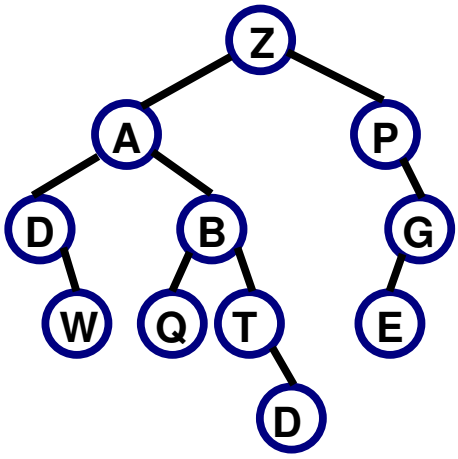
v) The second linked representation of Section 6.3.3 stores with each node a data value and a pointer to the list of children. Each child (every node except the root) has two additional pointers associated with it to indicate its place on the parent's linked list. Thus, the overhead fraction is

$$(3P) / (D + 3P)$$

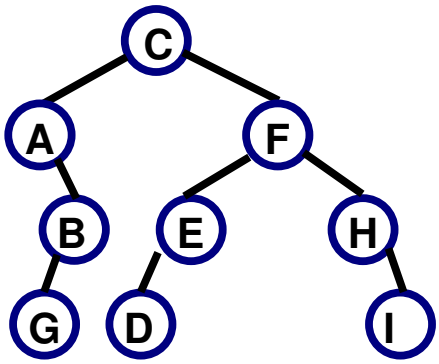
5. Draw the general tree represented by the following sequential representations:
   i) `ZAD/W//BQ//T/D//P/GE///`
   ii) `C'A'/B'G/F'E'D/H'/I`
   iii) `XPC)Q)RV)M))))`

Answer:

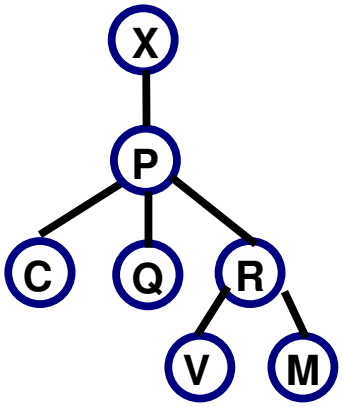
i)



ii)



iii)



6. Write a `converthelp` function to decode the sequential representation for binary trees illustrated by Q5.1. The input should be sequential representation and the output should be a pointer to the root of the resulting binary. Your function should have the following prototype:

```
template <class Elem>
BinNode<Elem>* convert(char* inlist) {
  int curr = 0;
  return converthelp(inlist, curr);
}
```

Answer:

```
// Use a helper function with a pass-by-reference
// variable to indicate current position in the
// node list.
template <class Elem>
BinNode<Elem>* convert(char* inlist) {
  int curr = 0;
  return converthelp(inlist, curr);
}

// As converthelp processes the node list, curr is
// incremented appropriately.
template <class Elem>
BinNode<Elem>* converthelp(char* inlist, int& curr) {
  if (inlist[curr] == '/') {
    curr++;
    return NULL;
  }
  BinNode<Elem>* temp = new BinNode(inlist[curr++],
                                    NULL, NULL);
  temp->left = converthelp(inlist, curr);
  temp->right = converthelp(inlist, curr);
  return temp;
}
```