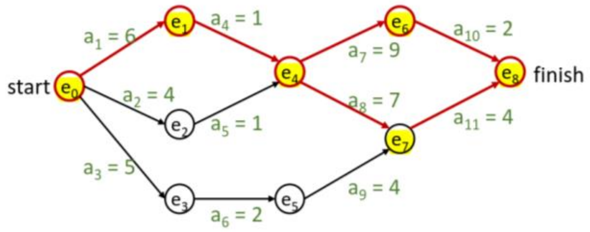
在critical path上，node的lastest time = earliest time

Critical Path

VoE

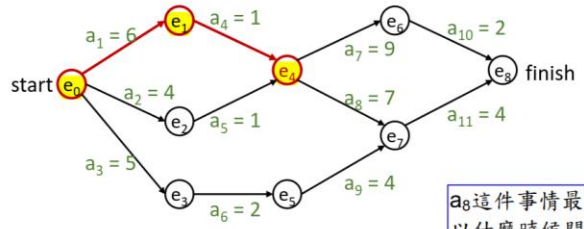
- The longest path from the start vertex to the finish vertex



- The above network has two critical paths
- Length(0, 1, 4, 6, 8) = 18
- Length(0, 1, 4, 7, 8) = 18

Earliest Event/Activity Time

- The length of the longest path from the start vertex to a vertex



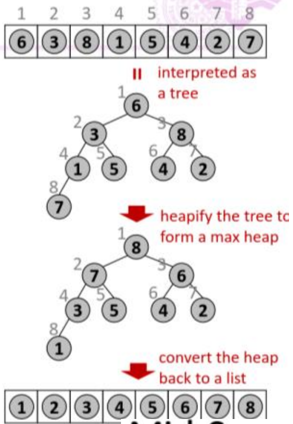
a_8這件事情最早可以什麼時候開始做

- Earliest event time(e_4) = 7 = 6 + 1
- Earliest activity time(a_7) = Earliest activity time(a_8) = 7
- Earliest event time(finish) = 18 = 6 + 1 + 11(9+2 or 7+4)

• Earliest event time(finish) = 18 • longest path length(e4, finish) = 11
• latest event time(e4) = 7 = 18 - 11

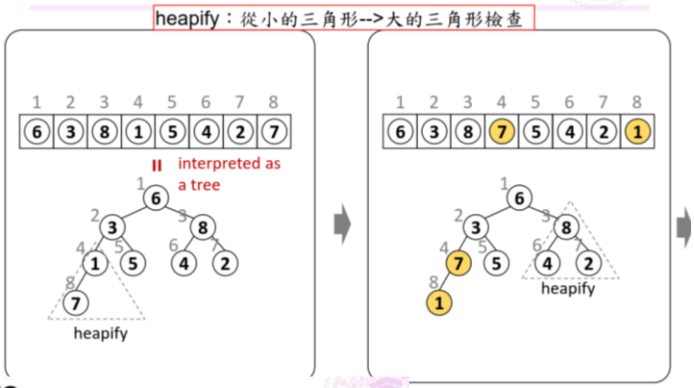
Heap Sort Concept

- Interpret the input list as a tree
- Heapify the tree to form a max heap
- Popping pass
  - Pop the top (maximum) record
    - Heap size shrinks by one
    - Space next to the heap becomes unused
  - Place the popped record at the space
- Popping passes are continued until the heap becomes empty
- Heap Sort is non-stable



Heap Sort Detail Steps

heapify：從小的三角形-->大的三角形檢查



Summary

	Worst	Average	
Insertion Sort	n^2	n^2	• Fastest method when n is small (e.g., n<100) • O(1) space • Stable
Quick Sort	n^2	nlogn	• Fastest method in practice • Require $O(n^2)$ time in the worst case • Require O(log(n)) space • Non-stable
Merge Sort	n·log(n)	n·log(n)	• Require additional O(n) space • Stable
Heap Sort	n·log(n)	n·log(n)	• Require additional O(1) space • Non-stable

Mid-Square

- h(k) = some middle r bits of the square of k
  - The number of bucket is equal to 2^r
- Example

k		k^2	h(k)
0	0	0000 0000	0
1	1	0000 0001	0
2	4	0000 0100	1
3	9	0000 1001	2
4	16	0001 0000	4
5	25	0001 1001	6
6	36	0010 0100	9
7	49	0011 0001	12

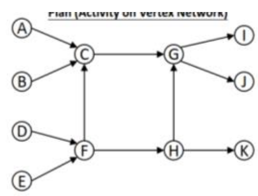
k		k^2	h(k)
8	64	0100 0000	0
9	81	0101 0001	4
10	100	0110 0100	9
11	121	0111 1001	14
12	144	1001 0000	4
13	169	1010 1001	10
14	196	1100 0100	1
15	225	1110 0001	8

Please perform decision tree based algorithm analyses.

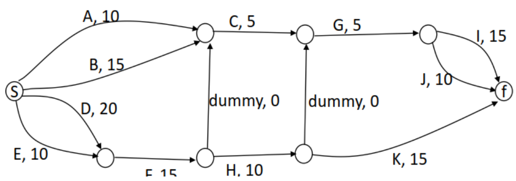
A. Please prove that any comparison-based algorithm requires log(N!) comparisons in the worst case to sort an N-element list. (5%)

Given N elements to sort, there are a total of N! possible outcomes. Therefore, the decision tree representation of any sorting algorithm must have N! outcomes, too. A decision tree of height k corresponds to at most 2^k outcomes. Therefore, the decision tree representation of sorting N elements must be at least of height log(N!). Any algorithm must perform log(N!) comparisons to sort N elements in the worst case.

Plan (Activity on Vertex Network)



Task	Required Time (Days)	Task	Required Time
A	10	G	5
B	15	H	10
C	5	I	15
D	20	J	10
E	10	K	15
F	15		



Folding

- Partition the key into several parts and add them together
  - Two strategies: shift folding and folding at the boundary
- Example
  - k = 12320324111220 = | 123 | 203 | 241 | 112 | 20 |
  - Shift folding

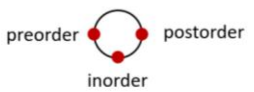
    $h(k) = \sum$ | 123 | 203 | 241 | 112 | 20 | = 699

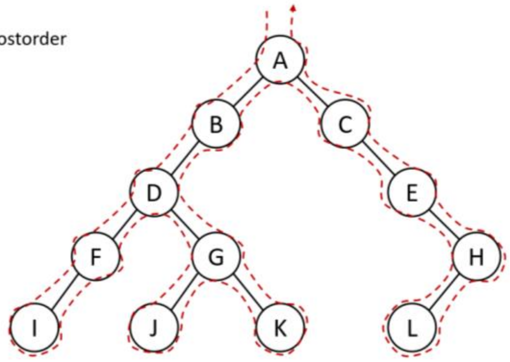
  - Folding at the boundary

    $h(k) = \sum$ | 123 | 302 | 241 | 211 | 20 | = 897

Tips for Preorder, Inorder, & Postorder

- Attach a point to each node
- Draw the contour of the tree





Searching a Binary Search Tree (Recursive)

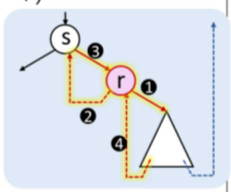
```
template <class K, class E>
pair<K, E>* BST<K, E> :: Get(const K& k)
{// Driver
    return rGet(root, k);
}

template <class K, class E>
pair<K, E>* BST<K, E> :: rGet(TreeNode <pair <K, E> >* p, const K& k)
{// Workhorse
   if (!p) return 0;
   if (k < p->data.first) return rGet(p->leftChild, k);
   if (k > p->data.first) return rGet(p->rightChild, k);
   return &p->data;
}
```

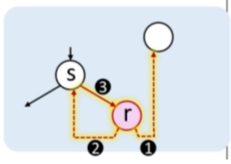
The two data members of an STL pair are named as "first" and "second"

It is correct to name the workhorse "Get" as the textbook does (because of function overloading). I change the name to "rGet" for clarity.

Inserting a Node into a Treaded Tree

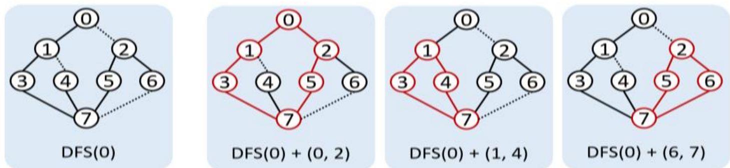
```
template <class T>
void ThreadedTree<T>::InsertRight(ThreadedNode <T> *s,
                                  ThreadedNode <T> *r)
{// insert r as the right child of s
    r->rightChild = s->rightChild;          ❶
    r->rightThread = s->rightThread;

    r->leftChild = s;                       ❷
    r->leftThread = true; //leftChild is a thread

    s->rightChild = r;                      ❸
    s->rightThread = false;

    if (! r->rightThread) {                 ❹
        ThreadedNode <T> *temp = InorderSucc(r);
        temp->leftChild = r;
    }
}
```



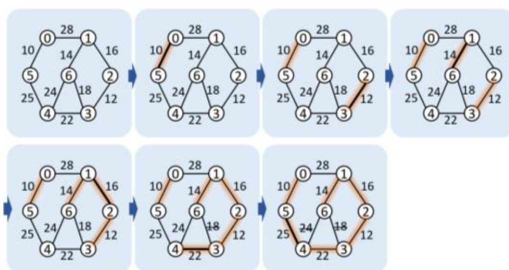


Spanning Tree → Independent Cycles

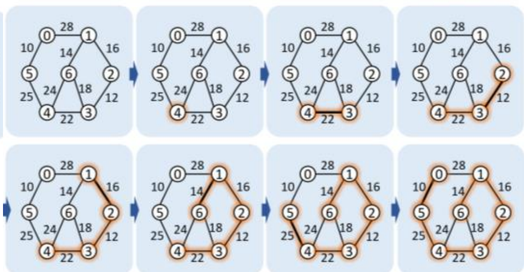
- Introducing a nontree edge (v, w) into a spanning tree produces a cycle
- These cycles are independent
  - Each introduced nontree edge is not contained in any other cycle
  - We cannot obtain any of these cycles by taking a linear combination of the remaining cycles
  - (# of independent cycles) = (# edges) - (# vertices - 1)



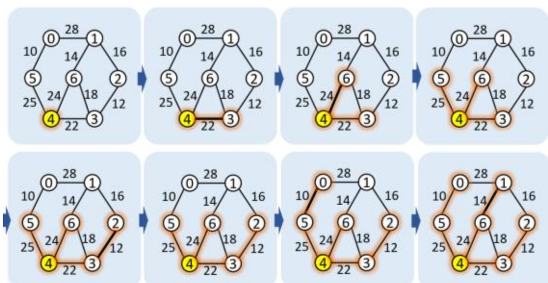
Kruskal's Example



Prim's Example

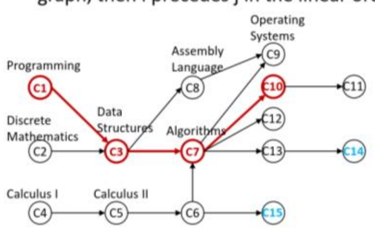


Dijastra's Example



Topological Order

- A linear order of the vertices of a graph such that
  - for any two vertices i and j, if i is a predecessor of j in the graph, then i precedes j in the linear ordering



Note:
- Transitivity among >2 vertices
- Topology order between two vertices does not always imply their precedence in the graph

- Two valid topological orderings (there are many of them)
  - **C1**, C2, C4, C5, **C3**, C6, C8, **C7**, **C10**, C13, C12, **C14**, **C15**, C11, C9
  - C4, C5, C2, **C1**, C6, **C3**, C8, **C15**, **C7**, C9, **C10**, C11, C12, C13, **C14**

Quick Sort Algorithm

```
template <class T>
void QuickSort(T *a, const int left, const int right)
{ // sort a[left..right]
    if (left < right) {
       int & pivot = a[left];
       int i = left;
       int j = right + 1;
       do {
          do j--; while (a[j] > pivot);   //find a key ≤pivot
          do { i++;                       //find a key >pivot
          } while (i < j && a[i] <= pivot);
          if (i < j) swap (a[i], a[j]);
       } while (i < j);
       swap (pivot, a[j]); //place the pivot between 2 lists
       QuickSort(a, left, j - 1);  // recursion
       QuickSort(a, j + 1, right); // recursion
    }
}
```