Computation IV — data structures and algorithms

1 Classification of Algorithms

Metrics: FLOPS Comparisons Interactions

Dependence of algorithm on inputs:

$O(1)$ constant time.

$O(\log N)$ Tree descent.

$O(N)$ Optimal if each datum must be processed.

$O(N \log N)$ Typical of divide and conquer algorithms.

$O(N^2)$ Quadratic: if every pair is examined.

$O(2^N)$ Exponential: if all permutations are examined.

2 Arrays

- Most fundamental data structure. (Why?)
- A primitive in most languages.
- Access time $O(1)$ if index is known.
- Insertion/deletion time $O(N)$.
- Multidimensional arrays.

  Either "vector of vectors" or matrix.

3 Lists

- A primitive in some languages (Lisp).
- Insertion/deletion time $O(1)$.
- Lookup time $O(N)$.
- can easily implement:
  1. Stacks: last in, last out storage.
  2. queues: first in, first out storage.
  3. deques: combination of stack and queue.

Implementing lists in arrays: indices can be used instead of links.

Example: Nearest Neighbor with chaining mesh

For each particle, find all particles within a distance L.

1. Divide space into cubical cells of size $\sim L$.
2. Allocate `LinkedList` array of size N_p; `Head` array of size N_{cell}. Initialize `Head` to NULLs.
3. Insert each particle into the linked lists:

   ```
   CellIndex = cell(particle);
   LinkedList[ParticleIndex] = head[CellIndex];
   head[CellIndex] = ParticleIndex;
   ```

4. The neighbors of a given particle can now be found by traversing the lists in `cell(particle)` and its neighbors.

4 Trees

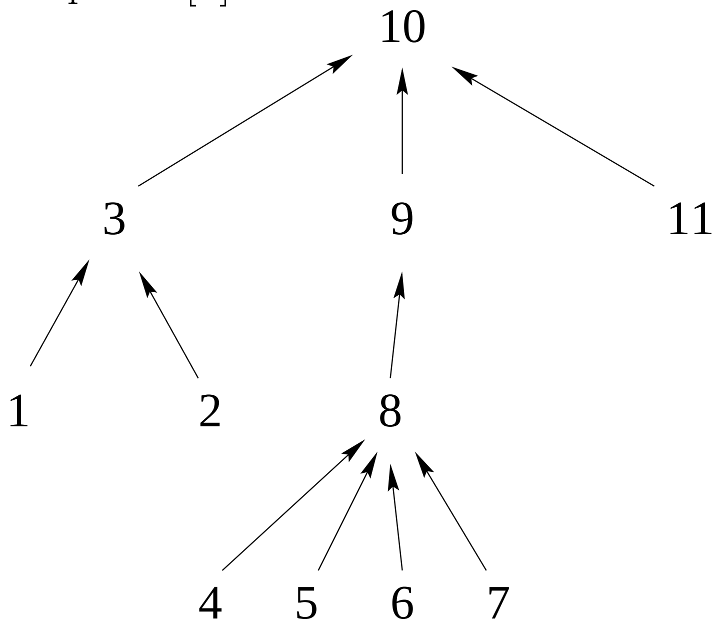
Glossary:

- node: Object that can carry information.
- edge: Connection between two nodes.
- leaf: Node with no children, also called external node.
- Lookup time $O(\log N)$.
- Insert/Delete time $O(\log N)$, but this can be tuned.

Trees with lists: parent trees.

Array representation:

k	1	2	3	4	5	6	7	8	9	10	11
parent[k]	3	3	10	8	8	8	8	9	10	0	10

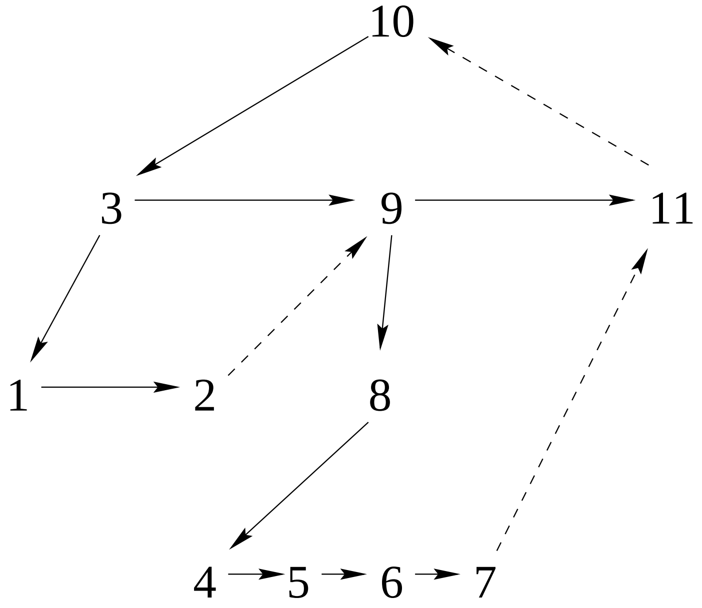


Example: Equivalence classes.

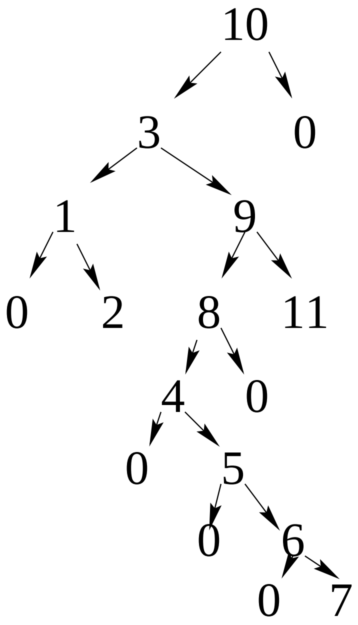
Given a set of objects, $A, B, C, \ldots$, and equivalence statements, $A \equiv C; \ldots$, separate the objects into distinct classes.

Represent each equivalence statement as a link, and build up a parent tree.

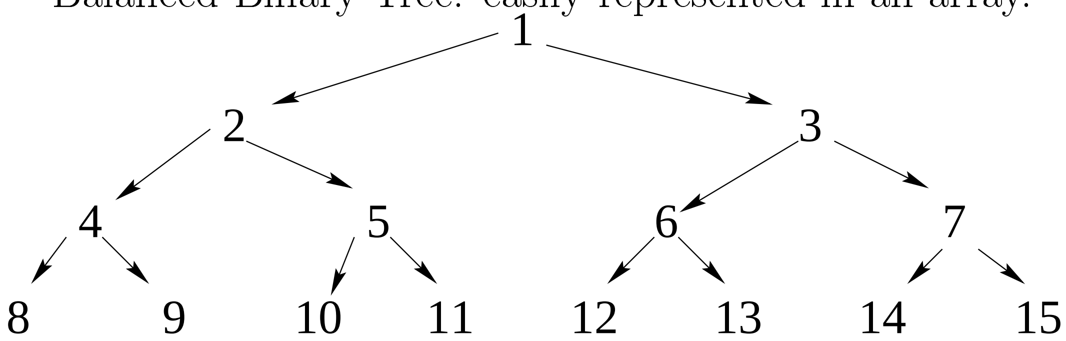
Child - sibling representation.



Binary Tree.



Balanced Binary Tree: easily represented in an array.



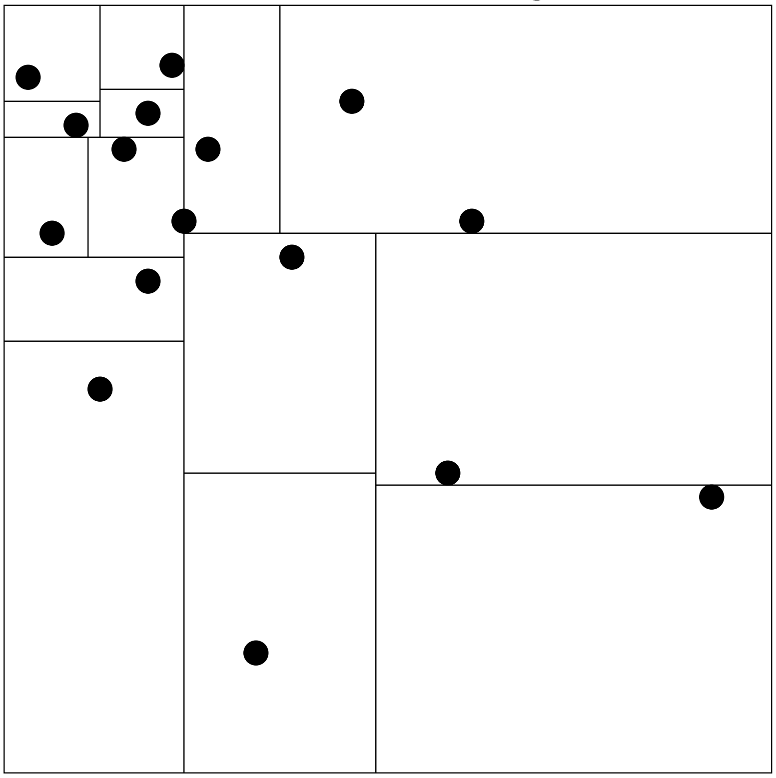
Depth first tree traversal:

```
Traverse(Node *t)
{
  if(!t) return;
  Traverse(t->left);
  DoNode(*t);
  Traverse(t->right);
}
```

Traversing a balanced binary tree:

```
{
  if((i << 1) < n)
    i <<= 1;
  else {
    while(i&1) i >>= 1;
    i++;
}
```

Example use: Nearest Neighbors with 2D tree.



5 Heaps

A heap is a balanced binary tree for which each node satisfies:

1. `n > n -> left`
2. `n > n -> right`

Two basic operations on a heap, both $O(\log N)$:

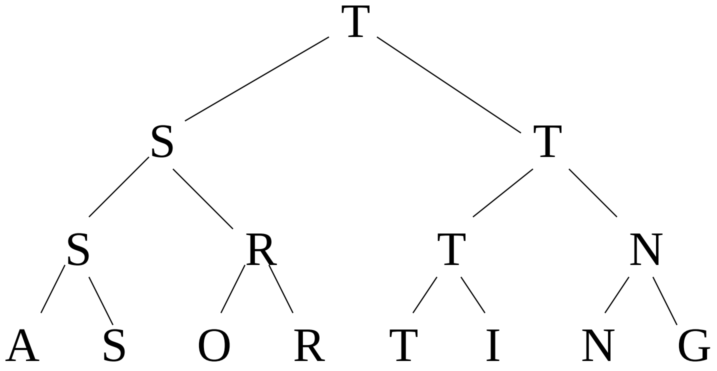
- insertion: place an element at the end of the heap and percolate it up.
- replace: put a new key at the root and percolate it downward.

Uses for a heap:

- Maintaining a priority queue.
- Sorting.

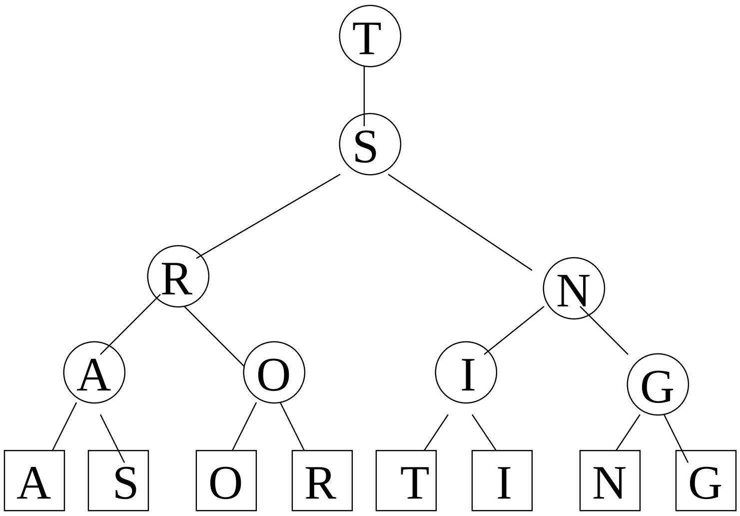
Selection Trees

All data is on external nodes.



The winner can be replaced by percolating it up the tree.
But the comparisons are all done with the previous losers.

Loser Trees



These are useful for priority queues if the size of the queue is aproximately constant.

6 Hashing

- Used for fast lookup: Is an item in my collection?
- A time-space trade-off.
  Use $O(N)$ memory to get $O(1)$ performance.

The hash function:

- Maps key into a table entry.
- Should be fast.
- Every output should be equally likely.

=> A random number generator.
$h(k) = kmodM$; M prime.

Collision resolution:
$h(k_1) = h(k_2)$ implies a collision.

1. Chaining
2. Linear probing
3. Double hashing

Comparison of Hashing with binary trees:

- Hashing is simpler.
- Hashing is fast.
- Hashing uses space.
- Trees are dynamic.
- Trees can have a guaranteed worst case performance.
- Trees support a wider range of operations.