IS 2610: Data Structures

Elementary Data Structures

Jan 26, 2004

First-In First Out Queues

- An ADT that comprises two basic operations: insert (put) a new item, and delete (get) the item that was least recently used

```
void QUEUEinit(int);
  int QUEUEempty();
void QUEUEput(Item);
Item QUEUEget();
```

```
typedef struct QUEUEnode* link;
struct QUEUEnode {Item item; link next;}
static link head;
link NEW(Item item, link next;}
   { link x = malloc(sizeof *x);
     x->item = item; x->next = next;
     return x;
   }
```

First-class ADT

- Clients use a single instance of STACK or QUEUE
- Only one object in a given program
- Could not declare variables or use it as an argument

- A first-class data type is one for which we can have potentially many different instances, and which can assign to variables whichcan declare to hold the instances

First-class data type – Complex numbers

- Complex numbers contains two parts
  - $(a + bi)$ where $i^2 = -1$;
  - $(a + bi)(c + di) = (ac - bd) + (ad + bc)i$

```
Typedef struct {float r; float i;} Complex;
Complex COMPLEXinit(float, float)
      float Re(float, float);
      float Im(float, float);
Complex COMPLEXmult(Complex, Complex)
```

```
Complex t, x, tx;
…
   t = COMPLEXInit(cos(r), sin(r))
   x = COMPLEXInit(?, ?)

  tx = COMPLEXmult(t, x)
```

First-class data type – Queues

```
typedef struct queue *Q;

void QUEUEdump(Q);
   Q QUEUEinit(int);
  int QUEUEempty(Q);
void QUEUEput(Q, Item);
Item QUEUEget(Q);
```

```
void QUEUEinit(int);
  int QUEUEempty();
void QUEUEput(Item);
Item QUEUEget();
```

```
Q queues[M];
for (i=0; i<M; i++)
   queues[i] = QUEUEinit(N);
.
   printf("%3d ", QUEUEget(queues[i]));
```

ADT

- ADTs are important software engineering tool
  - Many algorithms serve as implementations for fundamental
- ADTs encaptulate the algorithms that we develop, so that we can use the same code for many different applications
- ADTs provide a convenient mechanism for our use in the process of developing and comparing the performance of algorithms.

Recursion and Trees

- Recursive algorithm is one that solves a problem by solving one or more smaller instances of the same problem
  - Functions that call themselves
  - Can only solve a base case Recursive function calls itself
- If not base case
  - Break problem into smaller problem(s)
  - Launch new copy of function to work on the smaller problem (recursive call/recursive step)
    - Slowly converges towards base case
    - Function makes call to itself inside the return statement
  - Eventually base case gets solved
  - Answer works way back up, solves entire problem

Example of recursion

- Factorial of n: $n! = n*(n-1)*(n-2)*\ldots*1$
  - Recursive relationship ($n!=n*(n-1)!$)

    $5! = 5 * 4!$

    $4! = 4 * 3!\ldots$
  - Base case ($1! = 0! = 1$)
- Fibonacci number
  - Base case: $F_0 = F_1 = 1$
  - $F_n = F_{n-1} + F_{n-2}$

```
int factorial(int n){
    if (n=< 1) return 1;    //Base case
    return n*facorial(n-1);
}
```

```
int fibanacci(int n){
    ??; // Base Case
    return ??;
}
```

Euclid's algorithm
Greatest Common Divisor

- One of the oldest-known algorithm (over 2000 years)

Euclid's method for finding the greatest
Common divisor

```
int gcd(int m, int n){
        if (n==0) return m;
        return gcd(n, m%n);

}
```

```
56 ) 76 ( 1
     56
     20 ) 56 ( 2
          40
          16 ) 20 ( 1
               16
                4 ) 16 ( 4
                    16
                     0
```

Algorithm for pre-fix expression

```
char *a; int i;
int eval()
   {  int x = 0;
      while (a[i] == ' ') i++;
      if (a[i] == '+')
        { i++; return eval() + eval(); }
      if (a[i] == '*')
        { i++; return eval() * eval(); }
      while ((a[i] >= '0') && (a[i] <= '9'))
          x = 10*x + (a[i++]-'0');
      return x;
   }
```

```
eval () * + 7 6 12
    eval() + 7 6
        eval () 7
        eval () 6
        return 13 = 7 + 6
    eval () 12
    return 12 * 13
```

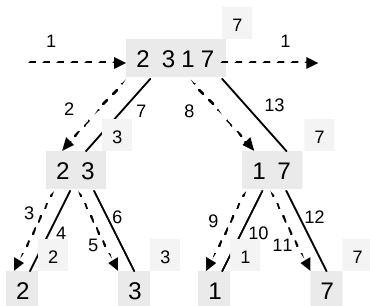
Recursive vs. iterative solution

- In principle, a loop can be replaced by an equivalent recursive program
  - Recursive program usually is more natural way to express computation
- Disadvantage
  - Nested function calls –
    - Use built in pushdown stack
    - Depth will depend on input
    - Hence programming environment has to maintain a stack that is proportional to the push down stack
    - Space complexity could be high

Divide and Conquer

- Many recursive programs use recursive calls on two subsets of inputs (two halves usually)
  - Divide the problem and solve them – divide and conquer paradigm
  - Property 5.1: a recursive function that divides a problem size N intro two independent (nonempty) parts that it solves recursively calls itself less than N times
  - Complexity: $T_N = T_k + T_{N-k} + 1$

Find max- Divide and Conquer




```
Item max(Item a[], int l, int r)
{  Item u, v;
   int m = (l+r)/2;
   if (l == r) return a[l];
   u = max(a, l, m);
   v = max(a, m+1, r);
   if (u > v) return u;
   else return v;
}
```

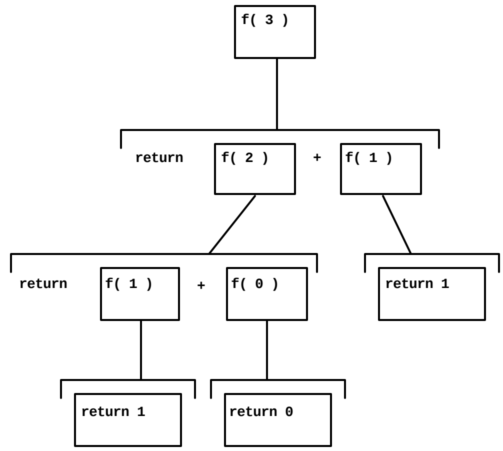
Dynamic programming

- When the sub-problems are not independent the situation may be complicated
  - Time complexity can be very high
- Example
  - Fibonacci number
    - Base case: $F_0 = F_1 = 1$
    - $F_n = F_{n\text{-}1} + F_{n\text{-}2}$

```
int fibanacci(int n){
    if (n=<1) return 1; // Base case
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Recursion: Fibonacci Series

- Order of operations
  - **return fibonacci(n - 1) + fibonacci(n - 2);**
- Recursive function calls
  - Each level of recursion doubles the number of function calls
    - 30th number = 2^30 ~ 4 billion function calls
  - Exponential complexity



Simpler Solution

- Linear!!
- Observation
  - We can evaluate any function by computing all the function values in order starting at the smallest, using previously computed values at each step to compute the current value
    - Bottom-up Dynamic programming
      - Applies to any recursive computation, *provided* that we can afford to save all the previously computed values
  - Top-down
    - Modify the recursive function to save the computed values and to allow checking these saved values
      - Memoization

```
F[0] = F[1] = 1;
For (i = 2; i<=N; i++);
   F[0] = F[i-1] + F[i-2];
```

Dynamic Programming

- Top-down : save known values
- Bottom-up : pre-compute values
  - Determining the order may be a challenge
- Top-down preferable
  - It is a mechanical transformation of natural problem
  - The order of computing the sub-problems takes care of itself
  - We may not need to compute answers to all the sub-problems

```
int F(int i)
{   int t;
    if (knownF[i] != unknown)
       return knownF[i];
    if (i == 0) t = 0;
    if (i == 1) t = 1;
    if (i > 1) t = F(i-1) + F(i-2);
      return knownF[i] = t;
}
```

Dynamic programming Knapsack problem

- *Property*: DP reduces the running times of a recursive function to be at most the time required to evaluate the function for all arguments less than or equal to the given argument
- Knapsack problem
  - Given
    - N types of items of varying size and value
    - One knapsack (belongs to a thief!)
  - Find: the combination of items that maximize the total value

Knapsack problem

Knapsack size: 17

	0	1	2	3	4
Item	A	B	C	D	E
Size	3	4	7	8	9
Val	4	5	10	11	13

```
int knap(int cap)
{  int i, space, max, t;
   for (i = 0, max = 0; i < N; i++)
      if ((space = cap - items[i].size) >= 0)
         if ((t = knap(space) + items[i].val) > max)
            max = t;
   return max;
}
```

```
int knap(int M)
{  int i, space, max, maxi, t;
   if (maxKnown[M] != unknown) return maxKnown[M];
   for (i = 0, max = 0; i < N; i++)
      if ((space = M-items[i].size) >= 0)
         if ((t = knap(space) + items[i].val) > max)  { max = t; maxi = i; }
   maxKnown[M] = max; itemKnown[M] = items[maxi];
return max; }
```

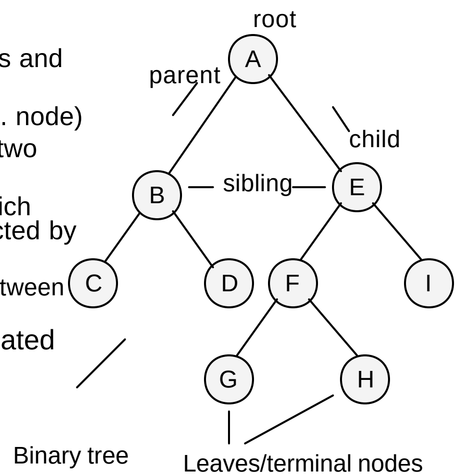
Tree

- Trees are central to design and analysis algorithms
  - Trees can be used to describe dynamic properties
  - We build and use explicit data structures that are concrete realization of trees

  General issues:
  - Trees
  - Rooted tree
  - Ordered trees
  - M-ary trees and binary trees

Tree

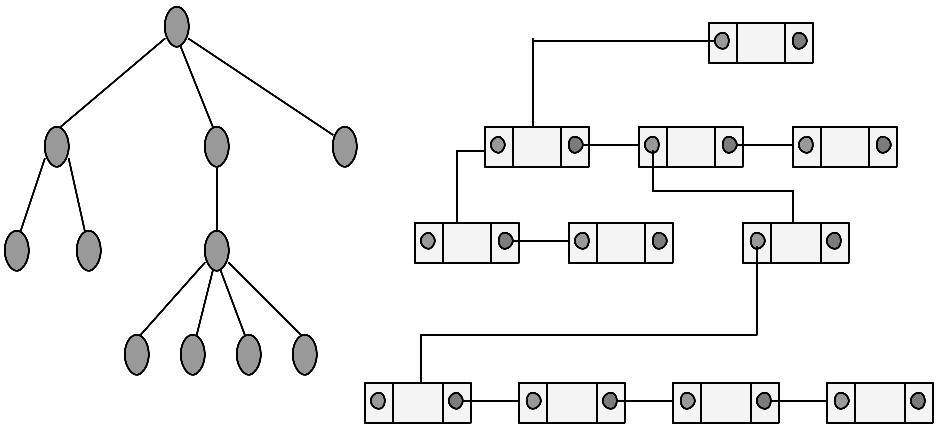
- Trees
  - Non-empty collection of vertices and edges
  - Vertex is a simple object (a.k.a. node)
  - Edge is a connection between two nodes
  - Path is a distinct vertices in which successive vertices are connected by edges
    - There is precisely one path between any two vertices
- Rooted tree: one node is designated as the root
- Forest
  - Disjoint set of trees



Definitions

- Binary tree is either an external node or an internal node connected to a pair of binary trees, which are called the left sub-tree and the right sub-tree of that node
  - Struct node {Item item; link left, link right;}
- M-ary tree is either an external node or an internal node connected to an ordered sequence of M-trees that are also M-ary trees
- A tree (or ordered tree) is a node (called the root) connected to a set of disjoint trees. Such a sequence is called a forest.
  - Arbitrary number of children
    - One for linked list connecting to its sibling
    - Other for connecting it to the sibling

Example general tree



Binary trees

- A binary tree with N internal nodes has N+ 1 external nodes
  - Proof by induction
  - $N = 0$ (no internal nodes) has one external node
  - Hypothesis: holds for N-1
  - k, N -1 - k internal nodes in left and right sub-trees (for k between 0 and N-1)
  - $(k+1) + (N - 1 - k) = N + 1$

Binary tree

- A binary tree with N internal nodes has 2N links
  - N-1 to internal nodes
    - Each internal node except root has a unique parent
    - Every edge connects to its parent
  - N+1 to external nodes
- Level, height, path
  - Level of a node is 1 + level of parent (Root is at level 0)
  - Height is the maximum of the levels of the tree's nodes
  - Path length is the sum of the levels of all the tree's nodes
  - Internal path length is the sum of the levels of all the internal nodes

Tree traversal (binary tree)

- Preorder
  - Visit a node,
  - Visit left subtree,
  - Visit right subtree
- Inorder
  - Visit left subtree,
  - Visit a node,
  - Visit right subtree
- Postorder
  - Visit left subtree,
  - Visit right subtree
  - Visit a node