Notes #2

Monday, February 17, 2020 12:55 AM

2/17/20 Applications of Stacks:

Runtime Stack

where are variables stored?
- Stack (everything else)
- Heap (new)
- Global (static)

2 + 3 infix

2 3 + postfix

(3*5+7*(1+4))*6 infix

3 5 * 7 1 4 + * + 6 * postfix

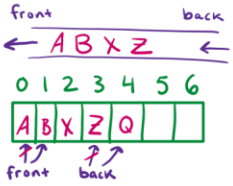
Evaluating PostFix

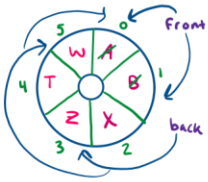
Scan left-to-right:

if it's a number
  Push it
if it's an operator
  Pop B
  Pop A
  Push (A op. B)

Queue:

- enqueue (insert)
- dequeue (remove)
- getFront - isfull
- size
- isEmpty - makeEmpty

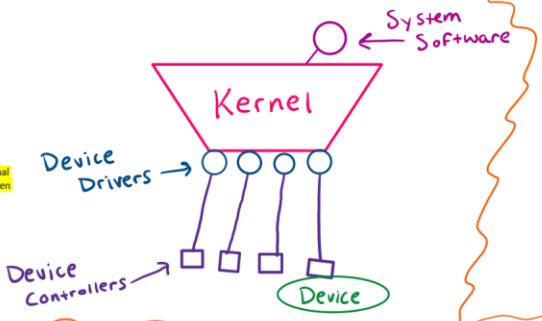




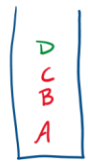
2/18/20 Operating Systems:

- Manages computer resources
  - Software packages
  - Memory Management → Virtual Memory & Paging
  - Storage (Disk) Management
  - Security / Protection
  - CPU Management
  - Process Management → A process is a program in execution

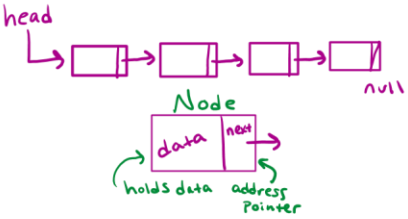
In this case the memory manager may create the illusion of additional memory space by rotating programs and data back and forth between main memory and mass storage (a technique called paging).

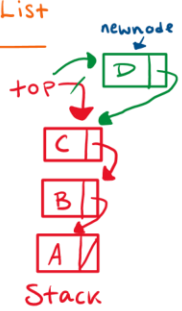


2/20/20 Stack using a Linked List



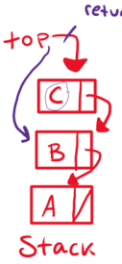
Push
Pop





Push D

1. Make a new empty node
2. Put D in the data part
3. Point the next pointer to the top ≡ the node holding C
4. Move top to point to the new node
5. Increment counter

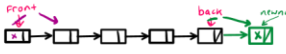


Pop C

1. Copy C out and Save it
2. Move top
3. Return the Saved item
5. Count --

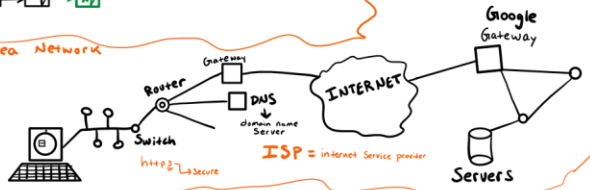
2/24/20 Queue:

enqueue
dequeue

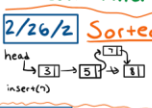


2/25/20 LAN → Local Area Network

- Ethernet:
  - bus-style broadcast network
- Sections are connected using:
  - Switch (smart)
  - Hub/repeater/bridge (dumb)
- Router (has a CPU in it)
  - Can Filter



2/26/2 Sorted List



Functions:

- insert (preserve order)
- get(i) Fetch item from position i
- delete(i)
- iterators

3/2/20 Recursion:

when a function calls itself

Rules:

1) Have a base case
2) Converge to a base case
3) Have faith (design rule)
4) Avoid repeated work

Dynamic Programming

- Using arrays to store function's values to avoid repeated work. (Rule 4)
- Usually the array is filled from 0

c(0) = 0 c(1) = 1 x is a coin

$c(n) = \min_{\text{coins } x} \{c(n-x)+1\}$

Recursion and Linked Lists

insert
delete
get

base case: pos = 0
rec. case: get from pos-1 from head.next

3/22/20 Big-Oh and Recursion

$t(0) = 2$
$t(n) = 3 + t(n-1)$

$t(n-1) = 3 + t(n-1-1) = 3 + t(n-2)$
$t(n-2) = 3 + t(n-3)$

$t(n) = K \cdot 3 + t(n-k)$
$K \le n$
$t(n) = 3n + t(0) = 3n + 2 = O(n)$

$t(0) = 2$
$t(n) = 4 + t(n/10)$
$t(n) = 4 + t(n/10^1)$
$t(n) = 4 + (4 + t(n/10^2))$
$= k \cdot 4 + t(n/10^k)$

$n/10^k < 1$
$n < 10^k$
$\log n < k \log 10$
$\frac{\log n}{\log 10} < k$

$K = O(\log n)$

$t(0) = 4$
$t(n) = 1 + t(n-1)$
$t(n-1) = 1 + t(n-2)$
$t(n) = 1 + (1 + t(n-2)) = 1 + (1 + 1 + t(n-3))$
$= k + t(n-k) = n + t(0) = n + 4 = O(n)$

Review/Quick Sort Analysis:

```
    break;
  }
 }
 if(i<0) list[offset]=item;
}
```

Analysis of Quicksort - Wednesday March 25

Sorts review:

Selection sort - O(n^2) - minimizes data moves
Bubble sort - O(n^2) - horrible!
Insertion sort - O(n^2) - good on nearly-sorted data
Shell sort - O(n (log n)^2) - clever and fast
Quicksort - O(n^2) in the worst case
- O(n log n) in the average (and best) case

Find max value move it to the end
Swap values
Insert values into sorted portion
Sort subsequences using gaps
Sort using divide and conquer & pivot

t(0)=t(1)=2
t(2)=7
t(n)=2+8n+8+t(L)+t(R)
L = left half
R = right half
t(n)=8n+10+t(k-1)+t(n-k)

Hand wavy

k is about n/2,
t(k-1) and t(n-k) are about t(n/2)

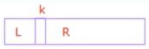
we get t(n) = 8n+10+2t(n/2) get O(n log n)

log n layers

if the pivot is at position k

t(n)=8n+10+t(k-1)+t(n-k)

t(n)=O(n)+t(n-1)



n t(n) = 8n+10+ (2/n) sum(k=1 to n) t(k-1)

(n-1)t(n-1) = 8(n-1) + 10 + (2/(n-1)) sum(k=1 to n-1) t(k-1)

nt(n) = 8n^2 + 10n + 2 sum(k=1 to n) t(k-1)

(n-1)t(n-1) = 8(n-1)^2 10(n-1) + 2 sum(k=1 to n-1) t(k-1)

n t(n) - (n-1)t(n-1) = 8(2n-1) + 10 + 2 t(n-1)
n t(n) = 16n +2 +(n+1)t(n-1)
divide through by n(n+1)
t(n)/(n+1) = 16/(n+1) + 2/(n(n+1)) + t(n-1)/n
s(n) = t(n)/(n+1) or (n+1)s(n)=t(n)
s(n) = 16/(n+1) + 2/(n(n+1)) + s(n-1)
s(n) = c/n + s(n-1) = c/n + c/(n-1)+s(n-2) ...
s(n) = c/n + c/(n-1) + c/(n-2)+ c/(n-3) + ...
) = O(log n)

$\sum \frac{1}{x}$

$\int \frac{1}{x}$

Programming Languages

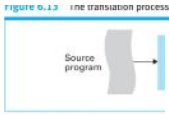
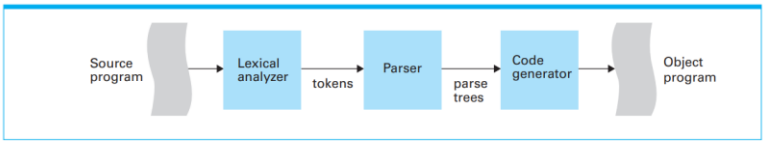


Figure 6.13 The translation process



Programming Languages - Brookshear ch. 6 - Thursday March 26

Programming Paradigms - ways of thinking

- Imperative - Fortran, COBOL, Pascal, Assembler/Machine, Basic, C
- Object-Oriented - Smalltalk, C++, Java
- Functional - LISP, Scheme, ML
- Declarative - PROLOG, [SQL?]

- Parallel/Concurrent - no good language [MPI]

Compiler Theory - program translation

- Lexical Analysis (Scanner) - produces a token stream
- Syntax Analysis (Parser) - checks the tokens against the grammar
- Semantic Analysis (Parser) - checks for meaning/types - create symbol table
- Code Generation
- Code Optimization

AWORD

Selection
ADORW
DIFERNT
DIFENRT
DEFINRT

Quicksort
AWORD

Bubble
AORDW
AODRW
ADORW

big=0 1
ADORW

Insertion
AWORD
AOWRD
AORWD
ADORW

Shell
AWORD
gap=2
AROWD

Merge
AOWDR
ADORW

```
class SNQ
{
  public static void main(String [] args)
  {
    Stack S=new StackLL();
    Queue Q=new QueueLL();

    S.push("V");
    Q.enqueue("P");
    S.push("N");
    Q.enqueue("D");
    S.push("A");
    Q.dequeue();

    while(!S.isEmpty())
      { Q.enqueue(S.pop()); }
    while(!Q.isEmpty())
      { System.out.print(Q.dequeue()); }

    Q.enqueue("R");
    S.push("S");
    S.push(Q.dequeue());
    Q.enqueue("E");

    while(!Q.isEmpty())
      { System.out.print(Q.dequeue()); }
    while(!S.isEmpty())
      { System.out.print(S.pop()); }

    System.out.println("\n");
  }
}
```

Given:

```
class node {
  public Comparable data;
  public node next;
}

boolean isHere(node head, Comparable x)
{
  // answer starts here

  // loopy answer

  node p=head;
  while(p!=null)          for(node p=head; p!=null; p=p.next)
  {
    if(p.data.compareTo(x)==0) return true;
    p=p.next;
  }
  return false;

  // recursive answer

  if(head==null) // empty list
    return false;
  else if(head.data.compareTo(x))
    return true;
  else
    return isHere(head.next,x);

}
```

```
maria(0) = 0
maria(1) = 5+maria(1-1) = 5
maria(2) = 3+maria(2-1) = 3+5 = 8
maria(3) = 4+maria(3-1) = 4+8 = 12
maria(4) = 5+maria(4-1) = 5+12 = 17
maria(5) = 3+maria(5-1) = 3+17 = 20
maria(6) = 4+maria(6-1) = 4+20 = 24

talos(1) = 'A'+maria(1) ='A'+5 = F
talos(5) = 'A'+20 = U
talos(4) = 'A'+17 = R
talos(6) = 'A'+24 = Y
```

```
class rec
{
  public static int maria(int x)
  {
    if(x==0) return 0;
    switch(x%3)
    {
      case 0:
        return 4+maria(x-1);
      case 1:
        return 5+maria(x-1);
      case 2:
        return 3+maria(x-1);
    }
    return 0;
  }

  public static void talos(int x)
  {
    System.out.print((char)('A'+maria(x)));
  }

  public static void main(String [] args)
  {
    talos(1); talos(5); talos(4); talos(6);
    System.out.println("");
  }
}
```

COVID LESSONS:

Sorting - Selection Sort Algorithm - Friday March 20

Problem: given a list of comparable items, rearrange them into non-decreasing order

6 algorithms:

Selection Sort - minimizes data moves
Bubble Sort - horrible
Insertion Sort - good on nearly-sorted data
Shell Sort - fast, based on insertion sort
Quicksort - recursive, optimal on average
Mergesort - recursive, optimal, needs extra space

Bubble Sort:

Sorting - Bubble Sort - Friday March 20th

Idea: swap adjacent items that are out of order

It's very slow, but very easy to code
When you find it, replace it with ANY OTHER SORT

8 1 7 2 6 3 5 4
1 7 2 6 3 5 4 8
1 2 6 3 5 4 7 8
1 2 3 5 4 6 7 8
1 2 3 4 5 6 7 8 ✔ ☺

Running Time: $O(n^2)$

```
public static void bubble(Comparable [] list)
 {
  for(int pass=0; pass<list.length; pass++)
  {
   boolean issorted=true;
   for(int i=0; i<list.length-1; i++)
    //if(list[i]>list[i+1]) swap(list,i,i+1);
    if(list[i].compareTo(list[i+1])>0)
    {
     swap(list,i,i+1);
     issorted=false;
    }
   if(issorted) return;
  }
 }
```

Selection Sort:

Selection Sort

8 1 7 2 6 3 5 4
4 1 7 2 6 3 5 8
4 1 5 2 6 3 7 8
4 1 5 2 3 6 7 8
4 1 3 2 5 6 7 8
2 1 3 4 5 6 7 8
2 1 3 4 5 6 7 8
1 2 3 4 5 6 7 8

repeat:
find the maximum
swap to the end

Running Time:
$O(n^2)$ - 2 nested for loops

Data moves:
O(n)

```
public static void selection(Comparable [] list)
 {
  //repeat
  // find the max
  // swap to the end
  for(int last=list.length-1; last>0; last--)
  {
   int maxpos=findmax(list,last);
   swap(list,maxpos,last);
  }
 }
private static void swap(Object [] list, int x, int y)
 {
  Object temp;
  temp=list[x];
  list[x]=list[y];
  list[y]=temp;
 }

private static int findmax(Comparable [] list, int last)
 {
  int maxpos=0;
  for(int i=1; i<=last; i++)
   //if(list[maxpos]<list[i]) maxpos=i;
   if(list[maxpos].compareTo(list[i])<0) maxpos=i;
  return maxpos;
 }
```

Insertion Sort:

Insertion Sort - Monday March 23rd

Idea - insert into an already-sorted portion of the array

8 1 7 2 6 3 5 4
1 8 7 2 6 3 5 4
1 7 8 2 6 3 5 4
1 2 7 8 6 3 5 4
1 2 6 7 8 3 5 4
1 2 3 6 7 8 5 4

Running time: O(n^2)

Can be very fast on almost-sorted lists.

```
public static void insertion(Comparable [] list, int offset, int gap)
 {
  for(int toinsert=offset+gap; toinsert<list.length; toinsert+=gap)
  {
   Comparable item=list[toinsert];
   int i;
   for(i=toinsert-gap; i>=0; i-=gap)
   {
    if(item.compareTo(list[i])<0)
    //if(item<list[i])
     list[i+gap]=list[i];
    else
    {
     list[i+gap]=item;
     break;
    }
   }
   if(i<0) list[offset]=item;
```

```
public static void insertion(Comparable [] list)
 {
  for(int toinsert=1; toinsert<list.length; toinsert++)
  {
   Comparable item=list[toinsert];
   int i;
   for(i=toinsert-1; i>=0; i--)
   {
    if(item.compareTo(list[i])<0)
    //if(item<list[i])
     list[i+1]=list[i];
    else
    {
     list[i+1]=item;
     break;
    }
   }
   if(i<0) list[0]=item;
/*
   // printing for illustration purposes
   System.out.println();
   for(int j=0; j<=toinsert; j++) System.out.println(list[j]);
*/
  }
 }
```

Review/Quick Sort Analysis:

Shell Sort:

Shell Sort - Monday March 23

Idea: sort subsequences of the array defined by offsets and gaps

25 1 24 2 23 3 22 4 21 5 20 6 19 7 18 8 17 9 16 10 15 11 14 12 13 — gap=5
3 1 4 2 5 8 6 9 7 10 15 11 14 12 13 20 17 19 16 18 25 22 24 21 23 — gap=3
2 1 4 3 5 7 6 9 8 10 12 11 14 15 13 16 17 19 20 18 21 22 24 25 22 — gap=1 (insertion sort)

Best known gap sequence: O(n (log n)^2)

gap = gap/2.2 (experimental)

*uses insertion

```
public static void shell(Comparable [] list)
 {
  int gap=list.length/5;
  while(gap>1)
  {
   for(int offset=0; offset<gap; offset++)
    insertion(list,offset,gap);
   gap=(int)(gap/2.2);
  }
  insertion(list); // gap=1 pass
 }
```

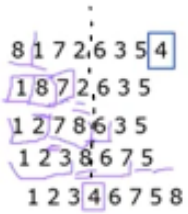
Quick Sort:

Quicksort - Tuesday March 24

Idea: recursive divide and conquer

*uses recursion

pivot



partition function

perform a partition
recursively sort left half
recursively sort right half

*its better to swap instead of slide

```
public static void quick(Comparable [] list)
 {
  quickhelp(list,0,list.length-1);
 }
private static void quickhelp(Comparable [] list, int start, int stop)
 {
  // base cases
  if(stop<=start) // length 0 or 1
   return;
  if(start+1==stop) // length 2
  {
   if(list[start].compareTo(list[stop])>0)
    swap(list,start,stop);
   return;
  }
// recursive case
  int pivotpos=partition(list,start,stop);
  quickhelp(list,start,pivotpos-1);
  quickhelp(list,pivotpos+1,stop);
 }
private static int partition(Comparable [] list, int start, int stop)
 {
  Comparable pivot=list[stop];
  int big=start;
  for(int i=start; i<stop; i++)
  {
   if(list[i].compareTo(pivot)<0)
   {
    swap(list,big,i);
    big++;
   }
  }
  swap(list,stop,big);
  return big;
```

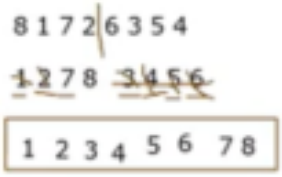
Avg Case: $O(n \log n)$

Worst Case: $O(n^2)$

Merge Sort:

Merge Sort - Friday March 27th

Idea: Sort each half, then merge
(recursive, divide and conquer)



Running Time:

merge: 9n+5

t(0)=t(1)=1 t(2)=4

t(n)= 9n+5+3 + 2t(n/2)

O(n log n)

```
public static void merge(Comparable [] list)
 {
  mergehelp(list,0,list.length-1);
 }
private static void mergehelp(Comparable [] list, int start, int stop)
 {
  // base cases
  if(stop<=start) // length 0 or 1
   return;
  if(start+1==stop) // length 2
  {
   if(list[start].compareTo(list[stop])>0)
    swap(list,start,stop);
   return;
  }
// recursive case
```