Sorting

CIS*2520 Summer 2006

Overview

- <u>Sorting</u>
  - induce some order on a set of values
- <u>Internal Sort</u>
  - sorting performed in main memory (most typical)
  - we will only consider internal sorting in this course
- <u>External Sort</u>
  - sorting performed in external storage (e.g. multiple tape merge)
- Recall
  - record containing key/data pair as a generic data element
  - equals, lessThan, greaterThan methods for generic comparison
  - we'll consider implementing these algorithms as operations on an ADT

Insertion Sort

- "Hand of Cards" analogy
  - look at cards one at a time, insert cards into correct location in (partially sorted) hand of cards
  - consider this operation on a row of cards on a table
    - as each new cards is seen it is compared with the row of cards and some are pushed one position to the right to make room for the new one
- General idea
  - keep first part of a list, once examined, in the correct order
  - an initial list with only one item is automatically in order
  - assuming we have already sorted the first i-1 items, we take item i and search the sorted list of length i-1 to see where to insert it
- Suitable for contiguous or linked implementations
  - contiguous: can use sequential or binary search to locate insertion position

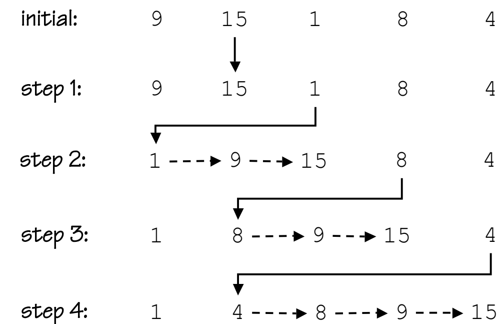
Insertion Sort Implementation

```
void insertionSort(List *l)
{
    int i,j;
    Record *tmp;

    for (int i = 1; i < l->item_count; i++)
    {
        tmp = l->items[i]; /* copy unsorted item */

        /* move items up to make room for item to be sorted */
        for (j = i-1; (j > 0)
                && (greaterThan(getKey(l->items(j-1)),
                getKey(tmp)) ; j--)
            l->items[j+1] = l->items[j];
    }
    /* copy unsorted item into correct location */
    l->items[j] = tmp;

} /* insertionSort */
```

Insertion Sort Implementation (cont.)

- list with only 1 item is automatically sorted so loop on *i* starts with *i == 1*
- Item is either already in correct position, or element to be sorted is copied and previous items in the list are copied up one position until the location for this item is found
- Linked version
  - left as an exercise
  - consider what additional record-keeping might be required or helpful
  - special cases?

Insertion Sort Example



Complexity of Insertion Sort

- Best Case
  - list already in order
  - n-1 iterations doing 1 comparison
  - $f(n) = n-1$ --- $f(n) \in \mathbf{O(n)}$
- Average Case
  - assume all orderings of keys are equally likely
  - for item *i*
    - probability key will not be moved (1 comparison) is $1/i$
    - probability key will be moved (1 comparison in outer loop, weighted summation over all possible locations with inner loop + assignments*)
    - derive expression for inner loop --- sum over all *i*
  - $f(n) = 1/4(n^2) + O(n)$ --- $f(n) \in \mathbf{O(n^2)}$
- Worst Case
  - left as an exercise (what is the worst case list?)

Notes on Insertion Sort

- Since a sorted list may incur arbitrary additional movements (read: assignment statements), assignments are significant to analysis
  - this relates to the next point as well...
- Problem
  - even after most items have been sorted insertion of a later item may require that many of them be moved
  - this is fine for small keys and records, but what about large items?

Selection Sort

- Similar "Hand of Cards" analogy
  - look over all the cards, select the highest one and place it where it belongs (at the end)
  - then select the second highest, etc.
- The general idea to selection sort is to select the item that belongs in a given location and move it directly into its final (sorted) position
- Minimizes data movement
  - best suited to contiguous representations (why?)

Selection Sort Implementation

```
void selectionSort(List *l)
{
    int i,j, max;
    Record *tmp;

    for (i = l->item_count - 1; i > 0; i--)
    {
        max = 0;
        for (int j = 1, j <= i; j++)
            if (lessThan(getKey(l->items[max]),getKey(l->items[j]))
                max = j;

        tmp = l->items[max];
        l->items[max] = l->items[i];
        l->items[i] = tmp;
    }

} /* selectionSort */
```

- Note: when all items but one are in the correct position, so is the last item

Selection Sort Example

initial:	9	15	1	8	4
step 1:	9	4	1	8	15
step 2:	8	4	1	9	15
step 3:	1	4	8	9	15
step 4:	1	4	8	9	15

Complexity of Selection Sort

- NOTE: # of comparisons performed is independent of initial order of list
  - we know exactly how many times each loop iterates
  - implications?
    - worse than insertionSort for nearly ordered lists, however worst case will differ little from best/avg; performance is predictable
- <u>Assignments</u>
  - if an item is already in final position, won't be moved; otherwise the resulting swap moves at least 1 into its final position
  - at most $n-1$ swaps are performed sorting a list of n items
  - $f(n) = 3n + O(1)$ assignments
    - contrast with $0.25n^2 + O(n)$ for insertion sort

Complexity of Selection Sort (cont.)

- Comparisons
  - inner loop called *n-1* times, with length of sub-list ranging from n down to 2
  - NOTE: if *m* is the size of this sublist, *m-1* comparisons are done to determine the maximum
  - $f(n) = 0.5n^2 + O(n)$ comparisons --- $f(n) \in \mathbf{O(n^2)}$
- Consider behaviour of insertion vs. selection sort as N grows
  - effect of moving items becomes a slow process?

Divide and Conquer

- Division of a problem into smaller but similar subproblems
  - compare with recursion
  - divide and conquer algorithms lend themselves well to recursive solutions
- Apply to sorting (think recursively!):
  - BASE CASE:
    - a list of size 1 is already sorted
  - INDUCTION STEP:
    - sort a list by dividing list in 2 and sorting each sub-list
- Some book-keeping is needed here somehow
  - how to recombine the sub-lists?

Divide and Conquer Sorting

- General idea:

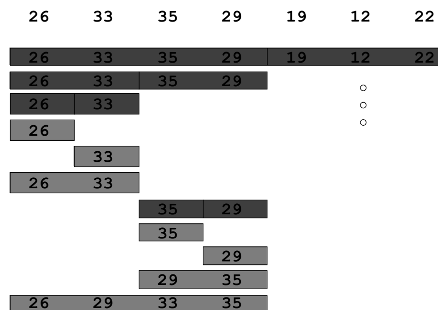
```
void sort()
{
    if list has length greater than 1
    {
        partition the list into lowlist, highlist
        sort(lowlist);
        sort(highlist);
        combine(lowlist,highlist);
    }

} /* sort */
```


Divide and Conquer Sorting (cont.)

- 2 approaches that we will consider:
  1. MERGESORT
     - chop the list into 2 sub-lists of sizes as nearly equal as possible
     - sort them separately
     - carefully merge the two sorted sub-lists into a single sorted list
  2. QUICKSORT
     - choose some key from the list for which we hope that approx. half the remaining keys will come before and half after (this key is the ***PIVOT***)
     - partition the items so that all those with keys less than the pivot come in one sub-list, and all those with greater keys come in another
     - sort the two reduced lists separately
     - put the sub-lists together resulting in a completely ordered list
  - note: quicksort does more work partitioning however the final step of combining the sub-lists becomes trivial

Mergesort Example

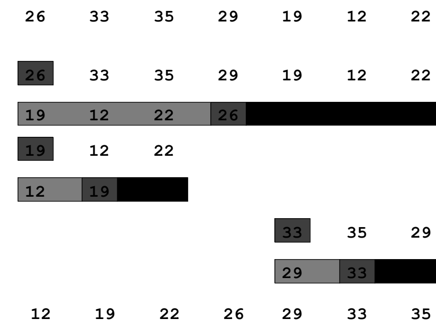
- Assume left sub-list is longer on partition when length is odd



CIS*2520 (S06)

Quicksort Example

- Assume chose first element in sub-list as pivot



CIS*2520 (S06)

Quicksort Implementation

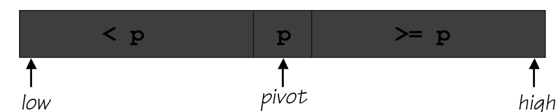
- Generally best for contiguous storage

```
void quickSort(List *l)
{
    qSort(l->items, 0, l->item_count-1);
} /* quickSort */


void _qSort(Record *items, int low, int high)
{
    int pivot;

    if (low < high)
    {
        pivot = partition(items,low,high);
        qSort(items,low,pivot-1);
        qSort(items, pivot+1,high);
    }
} /* _qSort */
```

CIS*2520 (S06)

Implementing Partition Operation

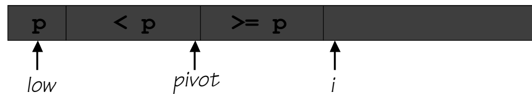
- In developing the "partition" function
  - given pivot value *p*, we must find a way to rearrange the entries of the array and compute the index "pivot" so that
    - the pivot is at this location
    - all entries to its left have keys less than *p* and all entries to its right have keys greater than or equal to *p*
  - i.e.



CIS*2520 (S06)

Implementing Partition Operation (cont.)

- Consider:
  - compare each key to the pivot (use a loop over *i*) - use "pivot" such that all items at or before it have keys less than *p* and suppose *p* is in the first position and we leave it there for the moment



- When the function inspects the key at *i* there are 2 cases:
  1. if entry is greater than or equal to *p* then *i* can be increased and array still has desired property
  2. if entry is less than *p*, the restore the property by increasing "pivot" and swapping that entry with entry *i*
  - at the end, we only need to swap the *pivot* from position *low* to position "pivot" to obtain the desired final arrangement

Choice of Pivot

- Many possibilities (some tradeoffs involved)
  - ***choose the first/last***
    - poor choice if list already sorted as one sublist will always be empty
  - ***choose one near the center***
    - hope that approx. 1/2 come before and after
  - ***chose one randomly***
    - this is a particularly good option
    - we can show the probability of nearly best case performance is extremely high
  - other possibilities
- We will choose one near the center

Partition Implementation

```
int partition(items, int low, int high)
{
    KeyType *pivotkey;
    int pivotloc;

    swap(items, low, (low+high) / 2);
    pivotkey = getKey(items[low]);
    pivotloc = low;

    for (i = low+1; i < high; i++)
        if (lessThan(getKey(items[i]), pivotkey)
        {
            pivotloc++;
            swap(items, pivotloc, i);
        }
     swap(items, low, pivotloc);

    return(pivotloc);
} /* partition */
```


Complexity of Quicksort

- Relies on solving recurrence relations
  - which we won't consider in depth at this time
- WORST CASE: ***$O(n^2)$***
  - e.g. 2 4 6 7 3 1 5
    - middle key is always largest in sub-list
  - still, this is a better choice than simply the first element
    - why?
- AVERAGE CASE: ***$O(n\log_2 n)$***
  - average behaviour of quicksort when applied to lists in random order turns out to be one of the best of any sorting method yet known
  - with a good technique for selecting a pivot, quicksort is generally quite effective and is ideally suited to contiguous memory
  - widely used

Mergesort Implementation

- Generally best for linked storage
  - we'll assume a non-circular single-linked list for illustration purposes

```
void mergeSort(List *l)
{
    l->head = _mSort(l->head);
} /* mergeSort */

Element *_mSort(Element *list)
{
    Record *sublist;

    if ((list != NULL) && (getNext(list) != NULL))
    {
        sublist = divide(list);
        list = _mSort(list);
        sublist = _mSort(sublist);
        return(_merge(list,sublist));
    }
} /* _mSort */
```


Divide Implementation

```
Element *divide(Element *list)
{
    Element *tmp1 = list;
    Element *tmp2 = getNext(getNext(tmp1));

    while (tmp2 != NULL)
    {
        tmp1 = getNext(tmp1);
        tmp2 = getNext(tmp2);
        if (tmp2 != NULL)
            tmp2 = getNext(tmp2);
    }

    tmp2 = getNext(tmp1); /* break list after tmp1 */
    setNext(tmp1, NULL);

    return(tmp2);

} /* divide */
```


Merge Implementation

```
Element *merge(Element *first, Element *second)
{
    Element *head, *tmp;

    if (lessThan(getKey(first),getKey(second))
    {
        head = first;
        first = getNext(first);
    }
    else
    {
        head = second;
        second = getNext(second);
    }

    tmp = head;  /* reference first entry of merged list */
```


Merge Implementation (cont.)

```
    while((first != NULL) && (second != NULL))
    {
        if (lessThan(getKey(first),getKey(second))
        {
            setNext(tmp,first);
            tmp = getNext(tmp);
            first = getNext(first);
        }
        else
        {
            setNext(tmp,second);
            tmp = getNext(tmp);
            second = getNext(second);
        }
    }
```

Merge Implementation (cont.)

```
    if (first != NULL)
        setNext(tmp,first);
    else
        setNext(tmp,second);

    /*
     * return new head of merged list
     */
    return(head);

} /* merge */
```


Complexity of Mergesort

- Comparison of keys done in only one place - within main loop of merge function
  - number of comparisons cannot exceed number of nodes being merged
  - assume *N* to be a power of 2 (i.e. $N = 2^M$ - either exact or a good approx.)
    - look at recursion tree of the algorithm
    - total length of sub-lists on each level is *N* (of course)
    - every item is treated in exactly one merge on each level
    - total comparisons done on each level cannot exceed *N*
    - for *N* a power of 2, tree is perfectly balanced - $ceil(log_2 n)$ depth
- Number of comparisons of keys done for list of size N is:
  - $f(n) < n * ceil(log_2 n)$ (this can be made more precise if desired)
  - $f(n) \in \mathbf{O(nlog_2 n)}$

Notes on Mergesort

- Best = Average = Worst case complexity (why?)
  - so why aren't we using Mergesort for everything?
- Consider:
  - relatively space efficient for linked storage
  - considerable effort expended finding the "middle" of the list
    - this can be addressed to some extent
  - very localized behaviour makes it ideal for applications to external sorting (i.e. blocks on a disk)
  - mergesort on contiguous storage tends to consume either space or time (or is complex)
    - ***heapsort*** is a better option for contiguous storage
    - see text for heapsort details (it is an important sort, but one we will not discuss in class)

Sorting Summary

- $O(n^2)$ sorts
  - insertion sort
  - selection sort
  - (technically bubble sort also, but who would care?)
- $O(nlog_2 n)$ sorts
  - quicksort (average case)
  - mergesort
  - heapsort (see lab)
- Remark:
  - best possible complexity for sort relying on comparison of keys is $O(nlog_2 n)$ --- proof omitted
  - is it possible to do better?
    - yes, but cannot use comparisons --- must rely on properties of the input
    - linear sorting (see lab)