CSCI 136: Data Structures and Advanced Programming

Lecture 33

Heapsort

Instructor: Dan Barowy

Williams

Topics

Heapsort

Which data structure?

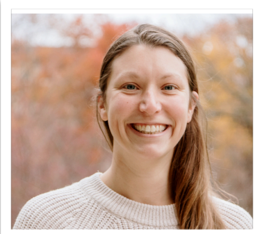
Announcements

1. **Final exam**: Saturday, Dec 17, 1:30pm. Room TBD.
2. **Final exam review session**, in class, last day of class, **Friday 12/9**.

Your to-dos

1. Lab 10 (partner lab), **due Tuesday 12/6 by 10pm**.
2. **Review readings** from *Bailey*.
3. **Study** for the final exam.
   a. Pro tip: **review quizzes**.
   b. **Do problems** in study guide/practice exam.
   c. **Don't stress out!** Just be methodical and do your best.
4. **Work on resubmissions** you plan to submit.

Announcements



Amy Babay, University of Pittsburgh

Friday, Dec 9 @ 2:35pm
Computer Science Colloquium – Wege TCL 123
Threats to Critical Infrastructure Control Systems

Amy runs the Resilient Systems and Societies Lab at the University of Pittsburgh School of Computing and Information, focusing on dependable infrastructure. The lab's work aims to make the networked systems our society relies on resilient to failures and attacks, and to develop new network technologies that help bring people together.

Amy conducts research on distributed systems and computer networks, aiming to not only fundamentally advance our understanding of how to design and build robust, performant systems that can be relied on to meet their requirements (despite failures and/or attacks), but also develop practical solutions with positive societal impact.

Heapsort

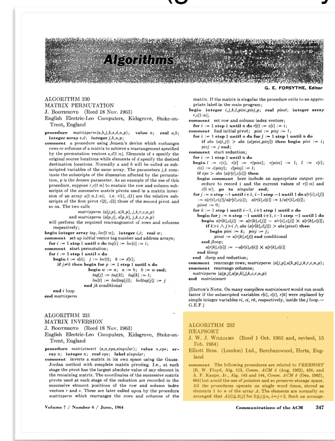
Heapsort

Heapsort is an **in-place, unstable sorting algorithm** that uses a binary min- or max-heap to order elements. The algorithm proceeds by first constructing a heap, then it removes elements one at a time to build a sorted array.

Like bubble sort, heapsort maintains the **invariant** that the rightmost **numSorted** elements are sorted. In other words, as elements are removed from the heap, they are inserted into the free space leftover after removal.

Heapsort

- Invented in 1963.
- Communications of the ACM generously gave the author two columns (glamorously titled "algorithm 232").



Algorithms

G. E. FORSYTHE, Editor

ALGORITHM 230
MATRIX PERMUTATION
J. Boothroyd (Recd 18 Nov. 1963)
English Electric-Leo Computers, Kidsgrove, Stoke-on-Trent, England

```
procedure matrixperm(a,b,j,k,s,d,n,p); value n; real a,b;
  integer array s,d; integer j,k,n,p;
comment a procedure using Jensen's device which exchanges
  rows or columns of a matrix to achieve a rearrangement specified
  by the permutation vectors s,d[1:n]. Elements of s specify the
  original source locations while elements of d specify the desired
  destination locations. Normally a and b will be called as sub-
  scripted variables of the same array. The parameters j,k nom-
  inate the subscripts of the dimension affected by the permuta-
  tion, p is the Jensen parameter. As an example of the use of this
  procedure, suppose r,c[1:n] to contain the row and column sub-
  scripts of the successive matrix pivots used in a matrix inver-
  sion of an array a[1:n,1:n]; i.e. r[1], c[1] are the relative sub-
  scripts of the first pivot r[2], c[2] those of the second pivot and
  so on. The two calls
        matrixperm (a[j,p], a[k,p], j,k,r,c,n,p)
    and matrixperm (a[p,j], a[p,k], j,k,c,r,n,p)
  will perform the required rearrangement of rows and columns
  respectively;
begin integer array tag, loc[1:n]; integer i,t; real w;
comment set up initial vector tag number and address arrays;
  for i := 1 step 1 until n do tag[i] := loc[i] := i;
comment start permutation;
  for i := 1 step 1 until n do
    begin t := s[i]; j := loc[t]; k := d[i];
      if j≠k then begin for p := 1 step 1 until n do
                  begin w := a; a := b; b := w end;
                  tag[j] := tag[k]; tag[k] := t;
                  loc[t] := loc[tag[j]]; loc[tag[j]] := j
                end jk conditional
    end i loop
end matrixperm
```

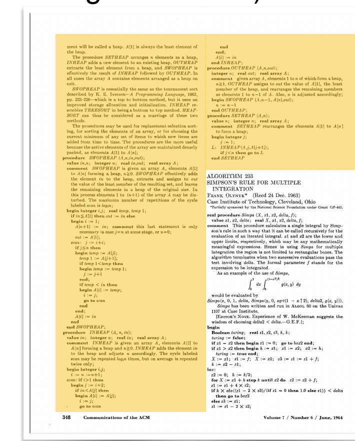
ALGORITHM 231
MATRIX INVERSION
J. Boothroyd (Recd 18 Nov. 1963)
English Electric-Leo Computers, Kidsgrove, Stoke-on-Trent, England

```
procedure matrixinvert (a,n,eps,singular); value n,eps; ar-
  ray a; integer n; real eps; label singular;
comment inverts a matrix in its own space using the Gauss-
  Jordan method with complete matrix pivoting. I.e., at each
  stage the pivot has the largest absolute value of any element in
  the remaining matrix. The coordinates of the successive matrix
  pivots used at each stage of the reduction are recorded in the
  successive element positions of the row and column index
  vectors r and c. These are later called upon by the procedure
  matrixperm which rearranges the rows and columns of the
  matrix. If the matrix is singular the procedure exits to an appro-
  priate label in the main program;
begin integer i,j,k,l,pivi,pivj,p; real pivot; integer array
  r,c[1:n];
comment set row and column index vectors;
  for i := 1 step 1 until n do r[i] := c[i] := i;
comment find initial pivot; pivi := pivj := 1;
  for i := 1 step 1 until n do for j := 1 step 1 until n do
    if abs (a[i,j]) > abs (a[pivi,pivj]) then begin pivi := i;
    pivj := j end;
comment start reduction;
  for i := 1 step 1 until n do
  begin l := r[i]; r[i] := r[pivi]; r[pivi] := l; l := c[i];
    c[i] := c[pivj]; c[pivj] := l;
    if eps > abs (a[r[i],c[i]]) then
    begin comment here include an appropriate output pro-
      cedure to record i and the current values of r[1:n] and
      c[1:n]; go to singular end;
  for j := n step -1 until i+1, i-1 step -1 until 1 do a[r[i],c[j]]
    := a[r[i],c[j]]/a[r[i],c[i]]; a[r[i],c[i]] := 1/a[r[i],c[i]];
  pivot := 0;
  for k := 1 step 1 until i-1, i+1 step 1 until n do
    begin for j := n step -1 until i+1, i-1 step -1 until 1 do
      begin a[r[k],c[j]] := a[r[k],c[j]] - a[r[i],c[j]] × a[r[k],c[i]];
        if k>i ∧ j>i ∧ abs (a[r[k],c[j]]) > abs(pivot) then
          begin pivi := k; pivj := j;
            pivot := a[r[k],c[j]] end conditional
      end jloop;
        a[r[k],c[i]] := -a[r[i],c[i]] × a[r[k],c[i]]
    end kloop
  end iloop and reduction;
comment rearrange rows; matrixperm (a[j,p],a[k,p],j,k,r,c,n,p);
comment rearrange columns;
  matrixperm (a[p,j],a[p,k],j,k,c,r,n,p)
end matrixinvert
```

[Editor's Note. On many compilers matrixinvert would run much faster if the subscripted variables r[i], c[i], r[k] were replaced by simple integer variables ri, ci, rk, respectively, inside the j loop.—G.E.F.]

ALGORITHM 232
HEAPSORT
J. W. J. Williams (Recd 1 Oct. 1963 and, revised, 15 Feb. 1964)
Elliott Bros. (London) Ltd., Borehamwood, Herts, England

```
comment The following procedures are related to TREESORT
  [R. W. Floyd, Alg. 113, Comm. ACM 5 (Aug. 1962), 434, and
  A. F. Kaupe, Jr., Alg. 143 and 144, Comm. ACM 5 (Dec. 1962),
  604] but avoid the use of pointers and so preserve storage space.
  All the procedures operate on single word items, stored as
  elements 1 to n of the array A. The elements are normally so
  arranged that A[i]≤A[j] for 2≤j≤n, i=j÷2. Such an arrange-
```

```
  ment will be called a heap. A[1] is always the least element of
  the heap.
    The procedure SETHEAP arranges n elements as a heap,
  INHEAP adds a new element to an existing heap, OUTHEAP
  extracts the least element from a heap, and SWOPHEAP is
  effectively the result of INHEAP followed by OUTHEAP. In
  all cases the array A contains elements arranged as a heap on
  exit.
    SWOPHEAP is essentially the same as the tournament sort
  described by K. E. Iverson—A Programming Language, 1962,
  pp. 223-226—which is a top to bottom method, but it uses an
  improved storage allocation and initialisation. INHEAP re-
  sembles TREESORT in being a bottom to top method. HEAP-
  SORT can thus be considered as a marriage of these two
  methods.
    The procedures may be used for replacement-selection sort-
  ing, for sorting the elements of an array, or for choosing the
  current minimum of any set of items to which new items are
  added from time to time. The procedures are the more useful
  because the active elements of the array are maintained densely
  packed, as elements A[1] to A[n];
procedure SWOPHEAP (A,n,in,out);
  value in,n; integer n; real in,out; real array A;
  comment SWOPHEAP is given an array A, elements A[1]
    to A[n] forming a heap, n≥0. SWOPHEAP effectively adds
    the element in to the heap, extracts and assigns to out
    the value of the least member of the resulting set, and leaves
    the remaining elements in a heap of the original size. In
    this process elements 1 to (n+1) of the array A may be dis-
    turbed. The maximum number of repetitions of the cycle
    labeled scan is log2n;
  begin integer i,j; real temp, temp 1;
    if in≤A[1] then out := in else
    begin i := 1;
      A[n+1] := in; comment this last statement is only
        necessary in case j=n at some stage, or n=0;
      out := A[1];
    scan: j := i+i;
      if j≤n then
      begin temp := A[j];
        temp 1 := A[j+1];
        if temp 1<temp then
        begin temp := temp 1;
          j := j+1
        end;
        if temp < in then
        begin A[i] := temp;
          i := j;
          go to scan
        end
      end;
      A[i] := in
    end
  end SWOPHEAP;
procedure INHEAP (A, n, in);
  value in; integer n; real in; real array A;
  comment INHEAP is given an array A, elements A[1] to
    A[n] forming a heap and n≥0. INHEAP adds the element in
    to the heap and adjusts n accordingly. The cycle labeled
    scan may be repeated log2n times, but on average is repeated
    twice only;
  begin integer i,j;
    i := n := n+1;
  scan: if i>1 then
    begin j := i÷2;
      if in<A[j] then
      begin A[i] := A[j];
        i := j;
        go to scan
      end
    end;
    A[i] := in
  end INHEAP;
procedure OUTHEAP (A,n,out);
  integer n; real out; real array A;
  comment given array A, elements 1 to n of which form a heap,
    n≥1, OUTHEAP assigns to out the value of A[1], the least
    member of the heap, and rearranges the remaining members
    as elements 1 to n-1 of A. Also, n is adjusted accordingly;
  begin SWOPHEAP (A,n-1, A[n],out);
    n := n-1
  end OUTHEAP;
procedure SETHEAP (A,n);
  value n; integer n; real array A;
  comment SETHEAP rearranges the elements A[1] to A[n]
    to form a heap;
  begin integer j;
      j := 1;
  L:  INHEAP(A,j,A[j+1]);
      if j<n then go to L
  end SETHEAP
```

ALGORITHM 233
SIMPSON'S RULE FOR MULTIPLE INTEGRATION
Frank Olynyk* (Recd 24 Dec. 1963)
Case Institute of Technology, Cleveland, Ohio
*Partially sponsored by the National Science Foundation under Grant GP-642.

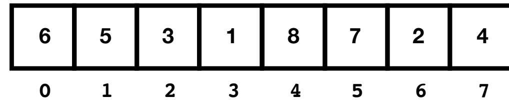
```
real procedure Simps (X, x1, x2, delta, f);
  value x1, x2, delta; real X, x1, x2, delta, f;
comment This procedure calculates a single integral by Simp-
  son's rule in such a way that it can be called recursively for the
  evaluation of an iterated integral. x1 and x2 are the lower and
  upper limits, respectively, which may be any mathematically
  meaningful expressions. Hence in using Simps for multiple
  integration the region is not limited to rectangular boxes. The
  algorithm terminates when two successive evaluations pass the
  test involving delta. The formal parameter f stands for the
  expression to be integrated.
    As an example of the use of Simps,
```

$$\int_0^1 dx \int_0^{(1-x^2)^{\frac{1}{2}}} g(x, y)\, dy$$

```
  would be evaluated by
Simps(x, 0, 1, delta, Simps(y, 0, sqrt(1 - x↑2), delta2, g(x, y))).
    Simps has been written and run in ALGOL 60 on the Univac
  1107 at Case Institute.
    [Editor's Note. Experience of W. McKeeman suggests the
  wisdom of choosing delta2 < delta.—G.E.F.];
begin
  Boolean turing; real z1, z2, z3, h, k;
  turing := false;
  if x1 = x2 then begin z1 := 0; go to box2 end;
  if x1 > x2 then begin h := x1; x1 := x2; x2 := h;
    turing := true end;
  X := x1; z1 := f; X := x2; z3 := z1 := z1 + f;
  k := x2 - x1;
box:
  z2 := 0; h := k/2;
  for X := x1 + h step k until x2 do z2 := z2 + f;
  z1 := z1 + 4 × z2;
  if h × abs((z1 - 2 × z3)/(if z1 = 0 then 1.0 else z1)) < delta
    then go to box2
  else z3 := z1;
  z1 := z1 - 2 × z2;
```

Heapsort: example

Suppose we have the following array of values,



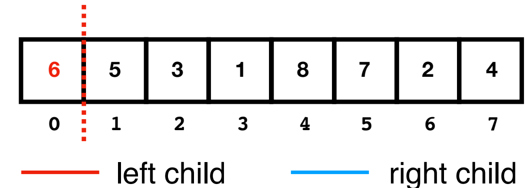
and that we want to produce an array in **ascending order** (i.e., from smallest to largest).

We first construct a **max heap** and then **remove** all the elements.

(If sorting in **descending order**, construct a **min heap**)

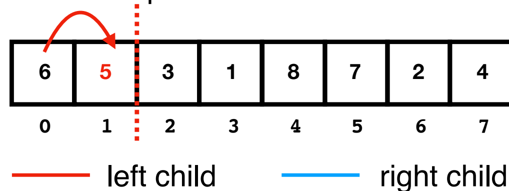
Heapsort: example

The first element is the new root.



Heapsort: example

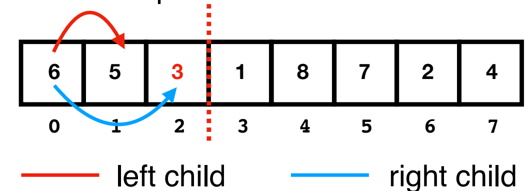
Add 5 to the max heap.



Since 5 is smaller than 6, it stays where it is (as the left child of 6).

Heapsort: example

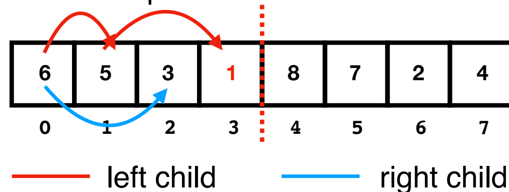
Add 3 to the max heap.



Since 3 is smaller than 6, it stays where it is (as the right child of 6).

Heapsort: example

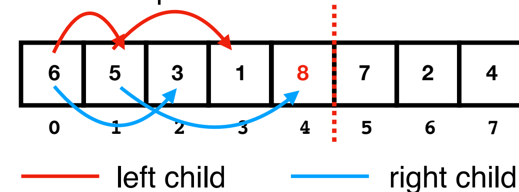
Add 1 to the max heap.



Since 1 is smaller than 5, it stays where it is (as the left child of 5).

Heapsort: example

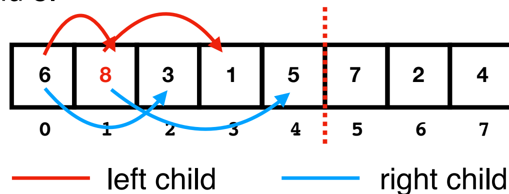
Add 8 to the max heap.



8 is not smaller than 5. So we run the "up heap" procedure.

Heapsort: example

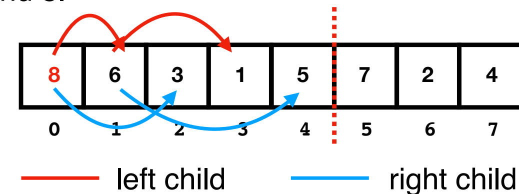
Swap 5 and 8.



8 is also not smaller than 6. So we run the "up heap" procedure.

Heapsort: example

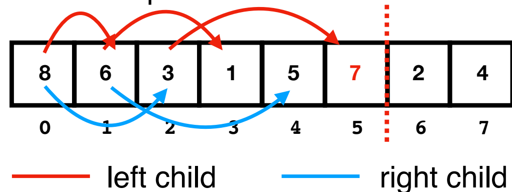
Swap 6 and 8.



Now 8 is in the correct location.

Heapsort: example

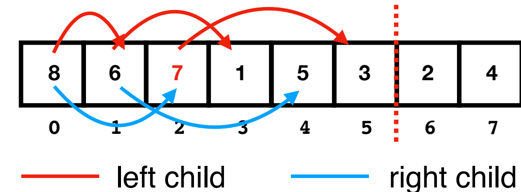
Add 7 to the max heap.



7 is not smaller than 3. So we run the “up heap” procedure.

Heapsort: example

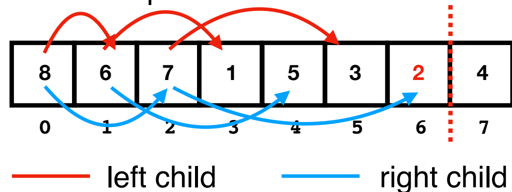
Swap 3 and 7.



7 is smaller than 8. So 7 is in the correct location.

Heapsort: example

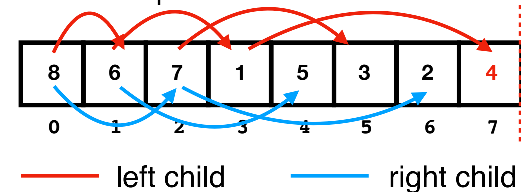
Add 2 to the max heap.



Since 2 is smaller than 7, it stays where it is (as the right child of 7).

Heapsort: example

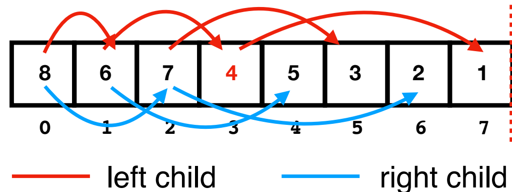
Add 4 to the max heap.



4 is not smaller than 1. So we run the “up heap” procedure.

Heapsort: example

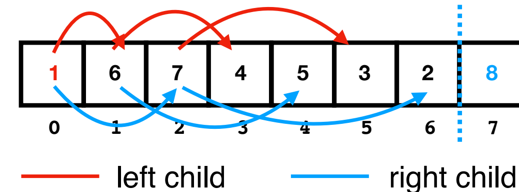
Swap 1 and 4.



4 is smaller than 6. So 4 is in the correct location.

Heapsort: example

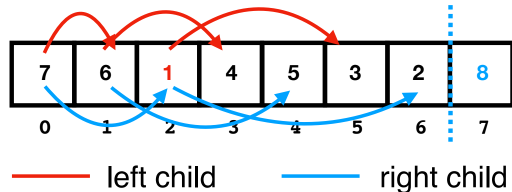
Extract the max by swapping it with the **size-1-numsortedth** element.



Since 1 is not bigger than 6 and 7, run the “down heap” procedure.

Heapsort: example

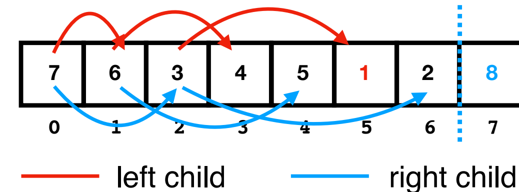
Swap 1 with 7.



Since 1 is not bigger than 3 and 2, run the “down heap” procedure.

Heapsort: example

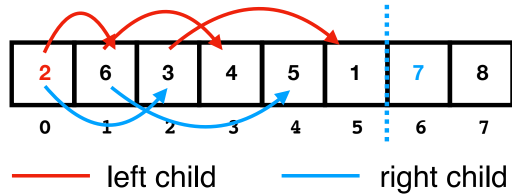
Swap 1 with 3.



Since 1 is now a leaf, we are done swapping.

Heapsort: example

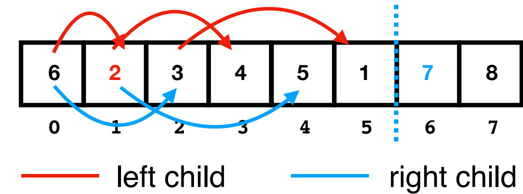
Extract the max by swapping it with the **size-1-numsorted**th element.



Since 2 is not bigger than 6 and 3, run the "down heap" procedure.

Heapsort: example

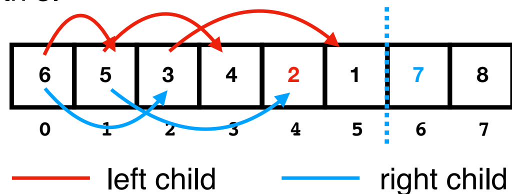
Swap 2 with 6.



Since 2 is not bigger than 4 and 5, run the "down heap" procedure.

Heapsort: example

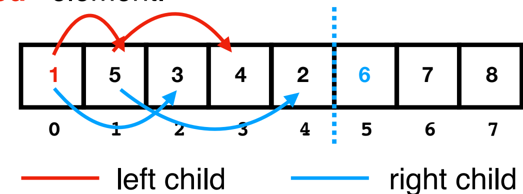
Swap 2 with 5.



Since 2 is now a leaf, we are done swapping.

Heapsort: example

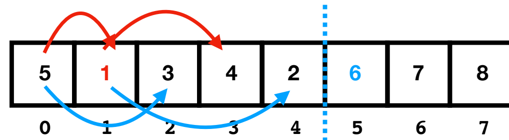
Extract the max by swapping it with the **size-1-numsorted**th element.



Since 1 is not bigger than 5 and 3, run the "down heap" procedure.

Heapsort: example

Swap 1 with 5.

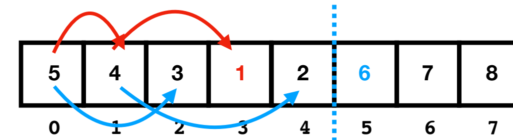


left child right child

Since 1 is not bigger than 4 and 2, run the “down heap” procedure.

Heapsort: example

Swap 1 with 4.

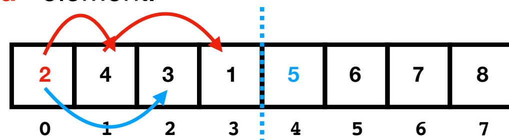


left child right child

Since 1 is now a leaf, we are done swapping.

Heapsort: example

Extract the max by swapping it with the **size-1-numsorted**th element.

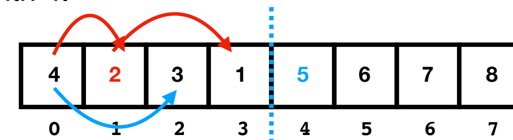


left child right child

Since 2 is not bigger than 4 and 3, run the “down heap” procedure.

Heapsort: example

Swap 2 with 4.

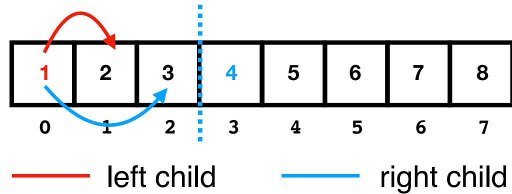


left child right child

Since 2 is bigger than 1, 2 is in the correct location.

Heapsort: example

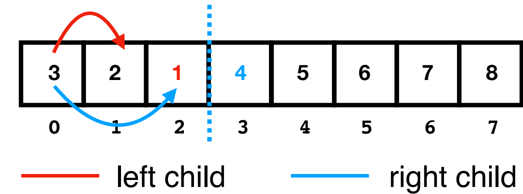
Extract the max by swapping it with the **size-1-numsorted**th element.



Since 1 is not bigger than 2 and 3, run the “down heap” procedure.

Heapsort: example

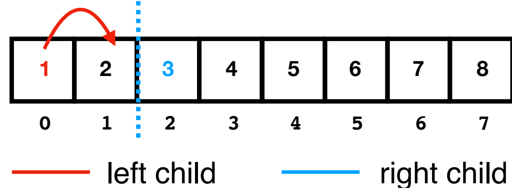
Swap 1 with 3.



Since 1 is now a leaf, we are done swapping.

Heapsort: example

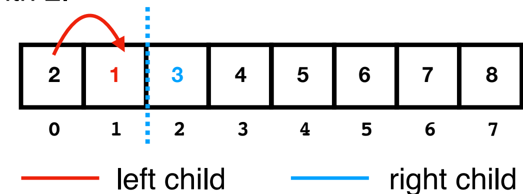
Extract the max by swapping it with the **size-1-numsorted**th element.



Since 1 is not bigger than 2, run the “down heap” procedure.

Heapsort: example

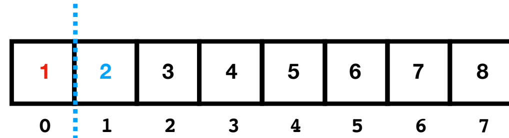
Swap 1 with 2.



Since 1 is now a leaf, we are done swapping.

Heapsort: example

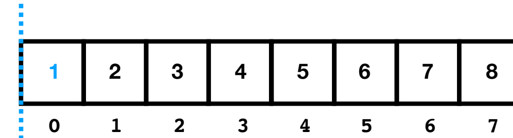
Extract the max by swapping it with the **size-1-numsorted**th element.



But since the heap only contains one element, it must be the max, so…

Heapsort: example

Done.



Activity:
Which data structure should I choose?

For each of the following **scenarios**, **any** of the data structures we've discussed this semester are possible choices.

However, not every data structure is a **good** choice.

Spend some time working with a partner and choose what you think is the **best data structure** for that **scenario**.

Justify your choice by **writing down the reasons** you chose it (hint: asymptotic arguments about space and time are good justifications!)

Activity:
Which data structure should I choose?

a.) You want to count occurrences of each word in a document then print an alphabetical list of word frequencies.

b.) You are writing code that will be used to store thousands of records (e.g. each record contains all the academic information for a specific student) and to retrieve them using a key (e.g. student name). The data rarely changes.

c.) Assume you are given a collection of pairs as input. Each pair contains the names of two people. Taken together, the pairs describe a social network. Over time, you will add pairs when friendships begin and remove them when friendships end. Data changes frequently. At any point in time, you want to be able to find the *k* most "popular" people.

d.) You want to count the occurrences of each letter in a document, then print an alphabetical list of letter frequencies.

Activity:

Which data structure should I choose?

a.) Two good choices come to mind: hash tables and binary search trees.
Hash table: O(n log n) = O(n) [to construct] + O(n log n) [to sort]
BST: O(n log n) = O(n log n) [to construct] + O(n) [to extract n sorted elements]

b.) A hash table is the clear winner: O(n) to construct, O(1) to do a lookup.

c.) The most obvious choice is a graph, but this one can have subtle issues.
Suppose we use a matrix to store it. We can add and remove edges in O(1). We can also maintain the top-k in O(k) as we add/remove; since k is fixed, it is a constant. However, a matrix may be wasteful of space, if the graph is sparse. For a sparse graph, an adjacency list would use less space, and since the graph is sparse maintaining lists may be relatively inexpensive. A hash table is another interesting option if a good hash function can be found for two people (e.g., hash(p1,p2) = p1 + p2), and sparseness is dealt with naturally by only storing entries where edges exist…

d.) Use an array! We know exactly how many characters there are in an alphabet, and if we restrict ourselves to a single language, that alphabet will be small (e.g., ~26, depending on what you want to count). Printing in alphabetical order is trivial, because the array stored character counts in alphabetical order. O(1) count, O(n) print.

Recap & Next Class

Today:

Heapsort

Which data structure should I choose?

Next class:

Dijkstra’s algorithm