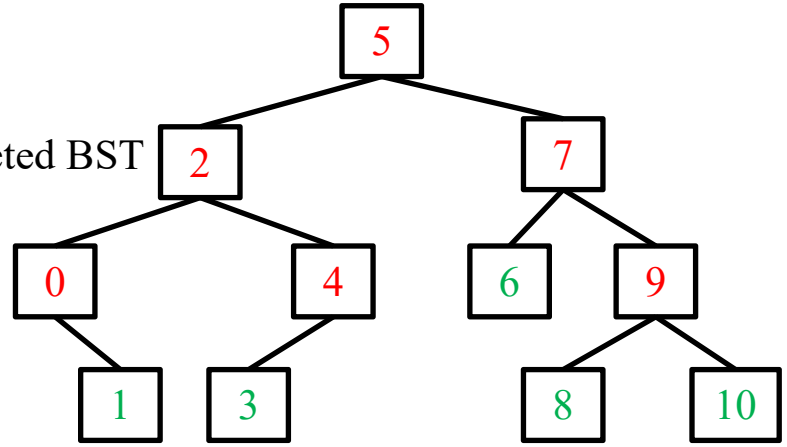
Sorting

- **BST (binary search tree) sort**
  - insert sequence into BST one by one
  - sort via inorder traversal of the completed BST



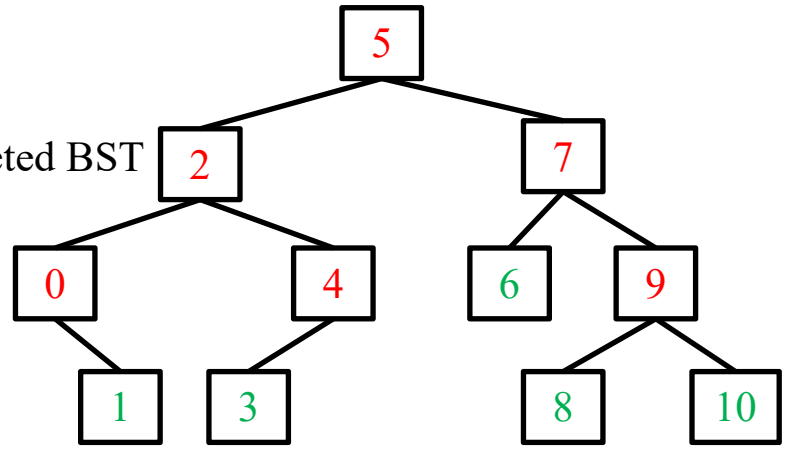
unsorted	5	2	4	3	7	6	0	1	9	10	8
inorder traversal	0	1	2	3	4	5	6	7	8	9	10

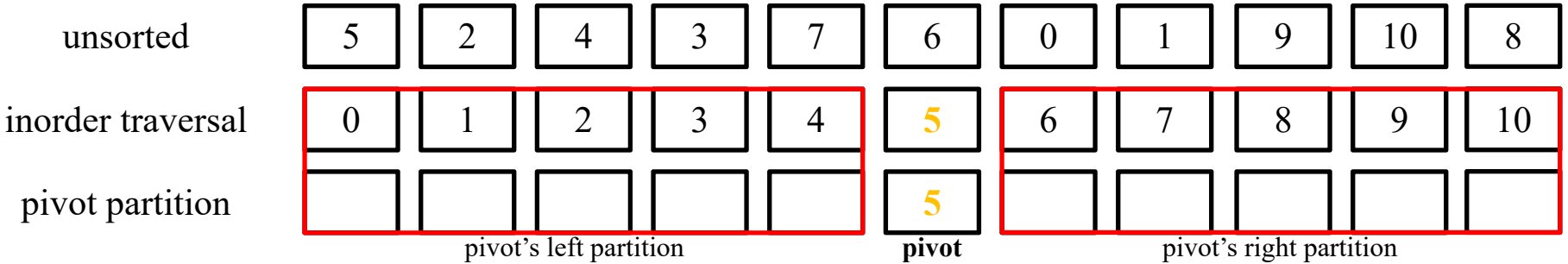
Sorting

- **BST (binary search tree) sort**
  - insert sequence into BST one by one
  - sort via inorder traversal of the completed BST
  - drawbacks of BST sort
    - extra space required by BST pointers
    - time required to insert (implying to create as well) nodes into BST
      - even aided by the *freelist* mechanism, creation needs time after all

Sorting

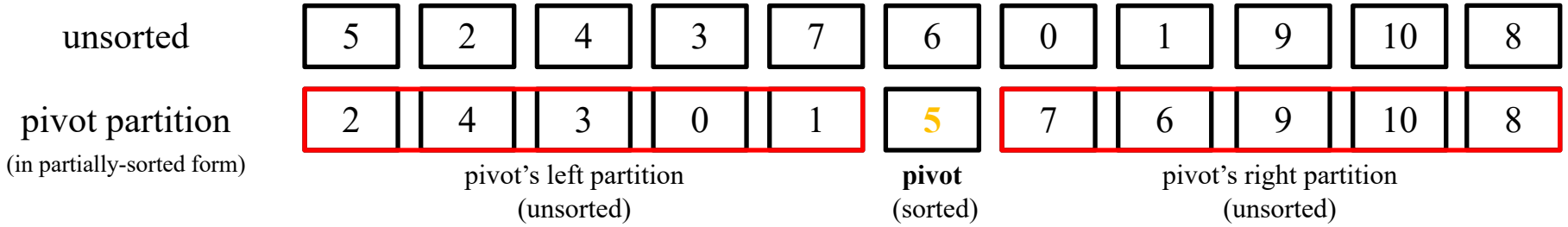
- **BST (binary search tree) sort**
  - insert sequence into BST one by one
  - sort via inorder traversal of the completed BST
  - drawbacks of BST sort
    - extra space required by BST pointers
    - time required to insert nodes into BST
  - idea of (root) *pivot partition*
    - reflects the spirit of *divide & conquer*





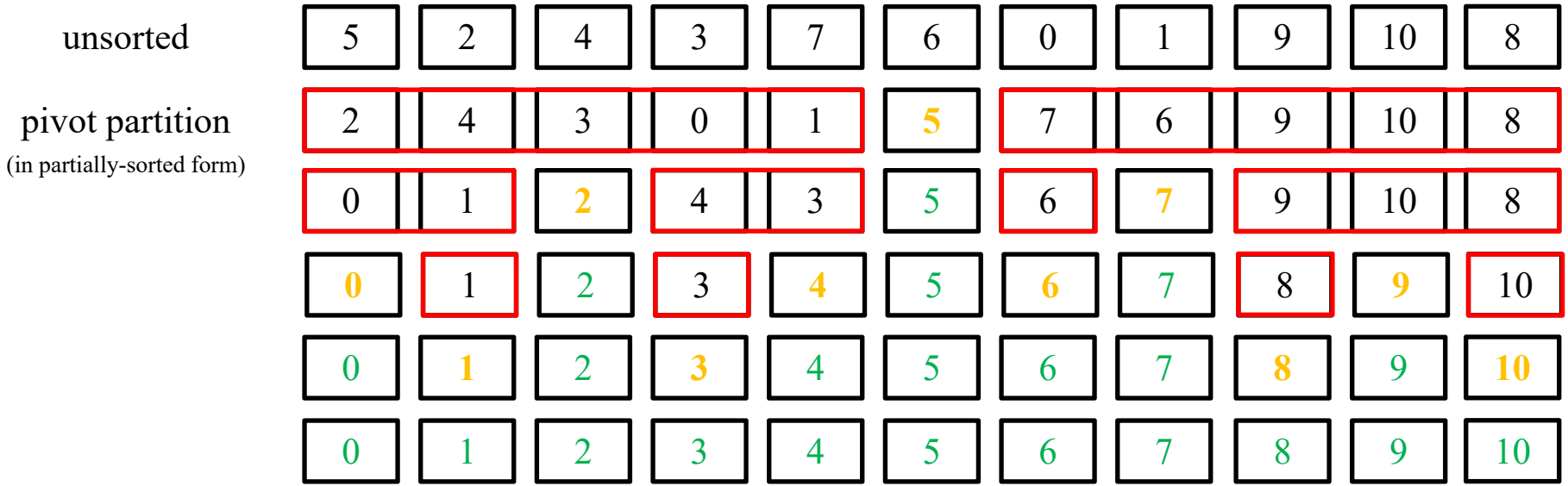
Sorting

- **Quicksort**
  - *divide & conquer*
    - divide into sub-tasks that are much easier and can be "merged" efficiently
  - *pivot partition in partially-sorted form (PSF)*
    - put sequence elements that are prior to the pivot to the left partition
    - put sequence elements to which the pivot is prior to the right partition
    - pivot is **sorted** namely its ordinal position is determined immediately
    - left & right partitions are **unsorted** and hence can be formed conveniently & quickly
    - unsorted left & right partitions are sorted via further **recursive pivot partition in PSF**



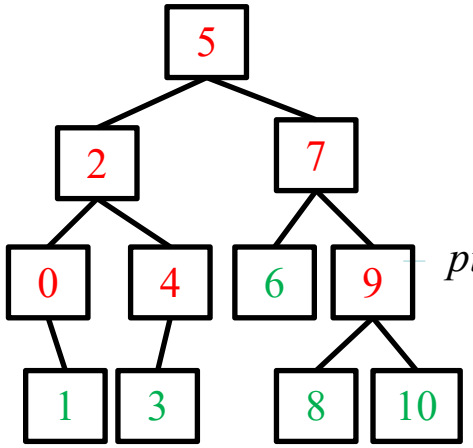
Sorting

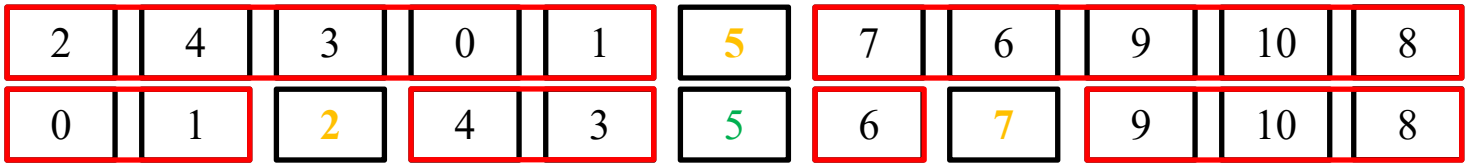
- **Quicksort**
  - *divide & conquer*
  - *pivot partition in partially-sorted form (PSF)*



Sorting

- **Quicksort - divide & conquer**
  - *pivot partition in partially-sorted form (PSF)*
    - put sequence elements that are prior to the pivot to the left partition
    - put sequence elements to which the pivot is prior to the right partition
    - pivot is **sorted** namely its ordinal position is determined immediately
    - left & right partitions are **unsorted** and hence can be formed conveniently & quickly
    - unsorted left & right partitions are sorted via further **recursive pivot partition in PSF**
  - *pivot partition in PSF* is what makes a difference between quicksort & BST sort
    - enable quicksort to be effective implemented on a single fixed array
    - exempt from dynamic maintenance of BST nodes (one-by-one insertion)
    - exempt from establishing unnecesary relations among sequence elements (extra pointers)





Sorting

- **Quicksort - divide & conquer**
  - *pivot partition in partially-sorted form (PSF)*
    - put sequence elements that are prior to the pivot to the left partition
    - put sequence elements to which the pivot is prior to the right partition
    - pivot is **sorted** namely its ordinal position is determined immediately
    - left & right partitions are **unsorted** and hence can be formed conveniently & quickly
    - unsorted left & right partitions are sorted via further **recursive pivot partition in PSF**
  - *pivot partition in PSF* is what makes a difference between quicksort & BST sort
  - complexity
    - worst case: $O(n^2)$
    - best case: O(n *log* n) $c(n) = 2c(\frac{n}{2}) + O(n)$
    - average case: O(n *log* n) $c(n) = \frac{1}{n-1}\sum_{k=1}^{n-1}[c(k) + c(n-k)] + O(n) = \frac{2}{n-1}\sum_{k=1}^{n-1} c(k)+O(n)$

Sorting

- **Quicksort - divide & conquer**
  - *pivot partition in partially-sorted form (PSF)*
    - put sequence elements that are prior to the pivot to the left partition
    - put sequence elements to which the pivot is prior to the right partition
    - pivot is **sorted** namely its ordinal position is determined immediately
    - left & right partitions are **unsorted** and hence can be formed conveniently & quickly
    - unsorted left & right partitions are sorted via further **recursive pivot partition in PSF**
  - *pivot partition in PSF* is what makes a difference between quicksort & BST sort
  - complexity - worst case $O(n^2)$; best & average cases $O(n \log n)$
  - *find pivot*
    - to select first or last tends to cause unbalanced partition (for nearly sorted or reverse sorted)
    - better to select a random element or expediently *array middle (good enough & efficient)*
    - even better to select "median of (random) three" or expediently median of first, middle, last

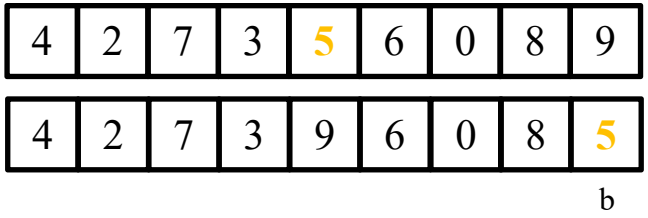
Sorting

- **Quicksort - divide & conquer**
  - *pivot partition in partially-sorted form (PSF)*

```
// quick sort: divide & conquer
#define QUICKSORT_SMALL_T 3
#define M_OF_3 0
template <class T> inline int findpivot(T s[],int iL,int iR){return (iL+iR-1)/2;}
template <class T,class P> inline int findpivot(T s[],int iL,int iR){
        int iM=(iL+iR-1)/2;if(P::p(s[iL],s[iM])==P::p(s[iM],s[iR-1])) return iM;
        else if(P::p(s[iM],s[iL])==P::p(s[iL],s[iR-1])) return iL;else return iR-1;}
template <class T,class P> void quicksort(T s[],int iL,int iR){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T) return; // do nothing for small sub-sequences
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a);quicksort<T,P>(s,a+1,iR);}
template <class T,class P> void quicksort(T s[],int iL,int iR,int d){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T || d<=0) return;
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a,d-1);quicksort<T,P>(s,a+1,iR,d-1);}
template <class T,class P> void quicksort(T s[],int n){
        quicksort<T,P>(s,0,n);
        // take advantage of the best-case performance of insertion sort by a
        // single final call to insertion sort to process the entire array
        insertionsort<T,P>(s,n);}
// END quick sort: divide & conquer
```

Sorting

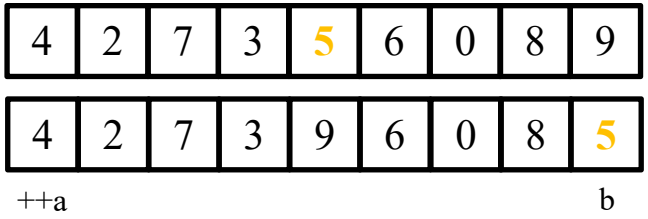
- **Quicksort - divide & conquer**
  - *pivot partition in partially-sorted form (PSF)*
  - partition trick




```
// quick sort: divide & conquer
#define QUICKSORT_SMALL_T 3
#define M_OF_3 0
template <class T> inline int findpivot(T s[],int iL,int iR){return (iL+iR-1)/2;}
template <class T,class P> inline int findpivot(T s[],int iL,int iR){
        int iM=(iL+iR-1)/2;if(P::p(s[iL],s[iM])==P::p(s[iM],s[iR-1])) return iM;
        else if(P::p(s[iM],s[iL])==P::p(s[iL],s[iR-1])) return iL;else return iR-1;}
template <class T,class P> void quicksort(T s[],int iL,int iR){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T) return; // do nothing for small sub-sequences
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a);quicksort<T,P>(s,a+1,iR);}
template <class T,class P> void quicksort(T s[],int iL,int iR,int d){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T || d<=0) return;
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a,d-1);quicksort<T,P>(s,a+1,iR,d-1);}
template <class T,class P> void quicksort(T s[],int n){
        quicksort<T,P>(s,0,n);
        // take advantage of the best-case performance of insertion sort by a
        // single final call to insertion sort to process the entire array
        insertionsort<T,P>(s,n);}
// END quick sort: divide & conquer
```

Sorting

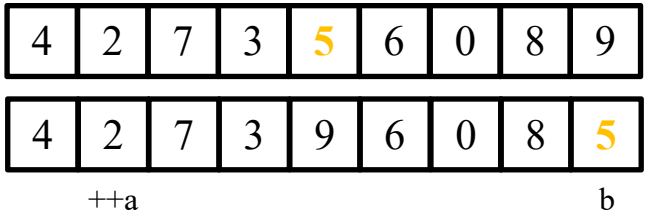
- **Quicksort - divide & conquer**
  - *pivot partition in partially-sorted form (PSF)*
  - partition trick




```
// quick sort: divide & conquer
#define QUICKSORT_SMALL_T 3
#define M_OF_3 0
template <class T> inline int findpivot(T s[],int iL,int iR){return (iL+iR-1)/2;}
template <class T,class P> inline int findpivot(T s[],int iL,int iR){
        int iM=(iL+iR-1)/2;if(P::p(s[iL],s[iM])==P::p(s[iM],s[iR-1])) return iM;
        else if(P::p(s[iM],s[iL])==P::p(s[iL],s[iR-1])) return iL;else return iR-1;}
template <class T,class P> void quicksort(T s[],int iL,int iR){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T) return; // do nothing for small sub-sequences
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a);quicksort<T,P>(s,a+1,iR);}
template <class T,class P> void quicksort(T s[],int iL,int iR,int d){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T || d<=0) return;
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a,d-1);quicksort<T,P>(s,a+1,iR,d-1);}
template <class T,class P> void quicksort(T s[],int n){
        quicksort<T,P>(s,0,n);
        // take advantage of the best-case performance of insertion sort by a
        // single final call to insertion sort to process the entire array
        insertionsort<T,P>(s,n);}
// END quick sort: divide & conquer
```

Sorting

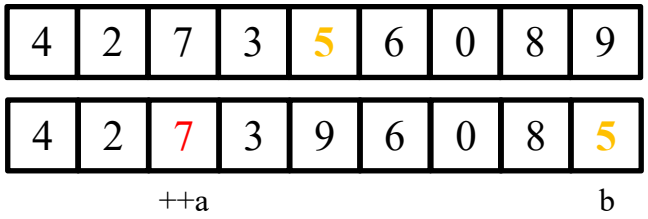
- **Quicksort - divide & conquer**
  - *pivot partition in partially-sorted form (PSF)*
  - partition trick




```
// quick sort: divide & conquer
#define QUICKSORT_SMALL_T 3
#define M_OF_3 0
template <class T> inline int findpivot(T s[],int iL,int iR){return (iL+iR-1)/2;}
template <class T,class P> inline int findpivot(T s[],int iL,int iR){
        int iM=(iL+iR-1)/2;if(P::p(s[iL],s[iM])==P::p(s[iM],s[iR-1])) return iM;
        else if(P::p(s[iM],s[iL])==P::p(s[iL],s[iR-1])) return iL;else return iR-1;}
template <class T,class P> void quicksort(T s[],int iL,int iR){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T) return; // do nothing for small sub-sequences
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a);quicksort<T,P>(s,a+1,iR);}
template <class T,class P> void quicksort(T s[],int iL,int iR,int d){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T || d<=0) return;
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a,d-1);quicksort<T,P>(s,a+1,iR,d-1);}
template <class T,class P> void quicksort(T s[],int n){
        quicksort<T,P>(s,0,n);
        // take advantage of the best-case performance of insertion sort by a
        // single final call to insertion sort to process the entire array
        insertionsort<T,P>(s,n);}
// END quick sort: divide & conquer
```

Sorting

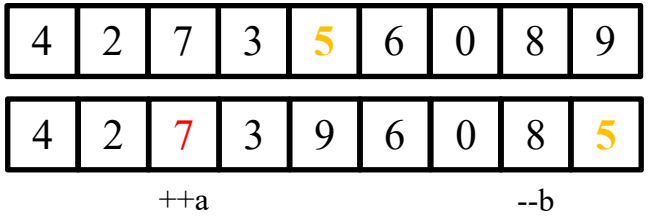
- **Quicksort - divide & conquer**
  - *pivot partition in partially-sorted form (PSF)*
  - partition trick




```
// quick sort: divide & conquer
#define QUICKSORT_SMALL_T 3
#define M_OF_3 0
template <class T> inline int findpivot(T s[],int iL,int iR){return (iL+iR-1)/2;}
template <class T,class P> inline int findpivot(T s[],int iL,int iR){
        int iM=(iL+iR-1)/2;if(P::p(s[iL],s[iM])==P::p(s[iM],s[iR-1])) return iM;
        else if(P::p(s[iM],s[iL])==P::p(s[iL],s[iR-1])) return iL;else return iR-1;}
template <class T,class P> void quicksort(T s[],int iL,int iR){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T) return; // do nothing for small sub-sequences
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a);quicksort<T,P>(s,a+1,iR);}
template <class T,class P> void quicksort(T s[],int iL,int iR,int d){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T || d<=0) return;
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a,d-1);quicksort<T,P>(s,a+1,iR,d-1);}
template <class T,class P> void quicksort(T s[],int n){
        quicksort<T,P>(s,0,n);
        // take advantage of the best-case performance of insertion sort by a
        // single final call to insertion sort to process the entire array
        insertionsort<T,P>(s,n);}
// END quick sort: divide & conquer
```

Sorting

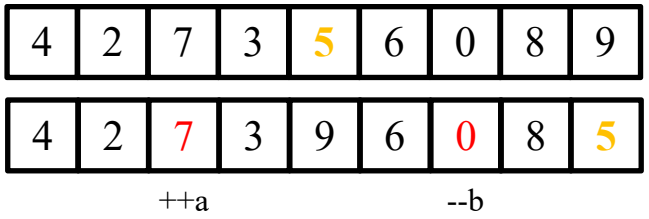
- **Quicksort - divide & conquer**
  - *pivot partition in partially-sorted form (PSF)*
  - partition trick




```
// quick sort: divide & conquer
#define QUICKSORT_SMALL_T 3
#define M_OF_3 0
template <class T> inline int findpivot(T s[],int iL,int iR){return (iL+iR-1)/2;}
template <class T,class P> inline int findpivot(T s[],int iL,int iR){
        int iM=(iL+iR-1)/2;if(P::p(s[iL],s[iM])==P::p(s[iM],s[iR-1])) return iM;
        else if(P::p(s[iM],s[iL])==P::p(s[iL],s[iR-1])) return iL;else return iR-1;}
template <class T,class P> void quicksort(T s[],int iL,int iR){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T) return; // do nothing for small sub-sequences
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a);quicksort<T,P>(s,a+1,iR);}
template <class T,class P> void quicksort(T s[],int iL,int iR,int d){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T || d<=0) return;
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a,d-1);quicksort<T,P>(s,a+1,iR,d-1);}
template <class T,class P> void quicksort(T s[],int n){
        quicksort<T,P>(s,0,n);
        // take advantage of the best-case performance of insertion sort by a
        // single final call to insertion sort to process the entire array
        insertionsort<T,P>(s,n);}
// END quick sort: divide & conquer
```

Sorting

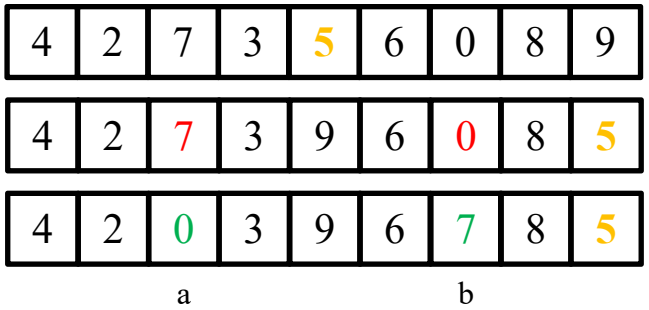
- **Quicksort - divide & conquer**
  - *pivot partition in partially-sorted form (PSF)*
  - partition trick




```
// quick sort: divide & conquer
#define QUICKSORT_SMALL_T 3
#define M_OF_3 0
template <class T> inline int findpivot(T s[],int iL,int iR){return (iL+iR-1)/2;}
template <class T,class P> inline int findpivot(T s[],int iL,int iR){
        int iM=(iL+iR-1)/2;if(P::p(s[iL],s[iM])==P::p(s[iM],s[iR-1])) return iM;
        else if(P::p(s[iM],s[iL])==P::p(s[iL],s[iR-1])) return iL;else return iR-1;}
template <class T,class P> void quicksort(T s[],int iL,int iR){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T) return; // do nothing for small sub-sequences
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a);quicksort<T,P>(s,a+1,iR);}
template <class T,class P> void quicksort(T s[],int iL,int iR,int d){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T || d<=0) return;
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a,d-1);quicksort<T,P>(s,a+1,iR,d-1);}
template <class T,class P> void quicksort(T s[],int n){
        quicksort<T,P>(s,0,n);
        // take advantage of the best-case performance of insertion sort by a
        // single final call to insertion sort to process the entire array
        insertionsort<T,P>(s,n);}
// END quick sort: divide & conquer
```

Sorting

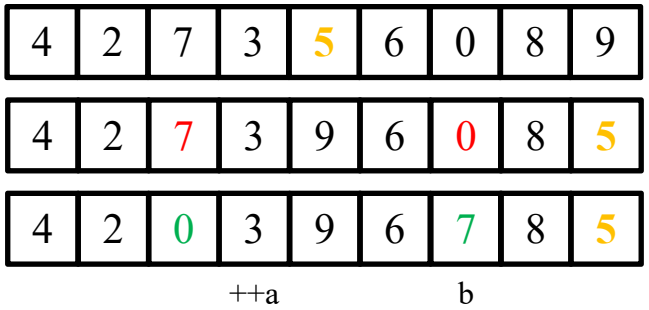
- **Quicksort - divide & conquer**
  - *pivot partition in partially-sorted form (PSF)*
  - partition trick




```
// quick sort: divide & conquer
#define QUICKSORT_SMALL_T 3
#define M_OF_3 0
template <class T> inline int findpivot(T s[],int iL,int iR){return (iL+iR-1)/2;}
template <class T,class P> inline int findpivot(T s[],int iL,int iR){
        int iM=(iL+iR-1)/2;if(P::p(s[iL],s[iM])==P::p(s[iM],s[iR-1])) return iM;
        else if(P::p(s[iM],s[iL])==P::p(s[iL],s[iR-1])) return iL;else return iR-1;}
template <class T,class P> void quicksort(T s[],int iL,int iR){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T) return; // do nothing for small sub-sequences
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a);quicksort<T,P>(s,a+1,iR);}
template <class T,class P> void quicksort(T s[],int iL,int iR,int d){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T || d<=0) return;
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a,d-1);quicksort<T,P>(s,a+1,iR,d-1);}
template <class T,class P> void quicksort(T s[],int n){
        quicksort<T,P>(s,0,n);
        // take advantage of the best-case performance of insertion sort by a
        // single final call to insertion sort to process the entire array
        insertionsort<T,P>(s,n);}
// END quick sort: divide & conquer
```

Sorting

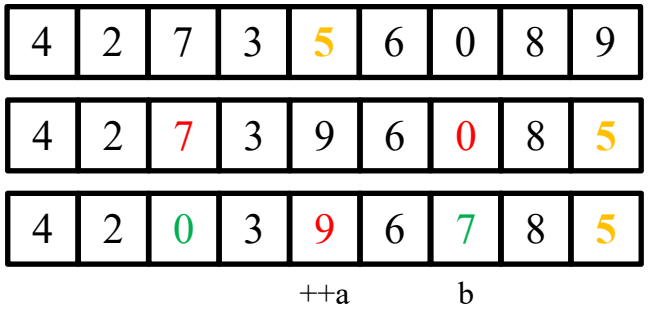
- **Quicksort - divide & conquer**
  - *pivot partition in partially-sorted form (PSF)*
  - partition trick




```
// quick sort: divide & conquer
#define QUICKSORT_SMALL_T 3
#define M_OF_3 0
template <class T> inline int findpivot(T s[],int iL,int iR){return (iL+iR-1)/2;}
template <class T,class P> inline int findpivot(T s[],int iL,int iR){
        int iM=(iL+iR-1)/2;if(P::p(s[iL],s[iM])==P::p(s[iM],s[iR-1])) return iM;
        else if(P::p(s[iM],s[iL])==P::p(s[iL],s[iR-1])) return iL;else return iR-1;}
template <class T,class P> void quicksort(T s[],int iL,int iR){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T) return; // do nothing for small sub-sequences
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a);quicksort<T,P>(s,a+1,iR);}
template <class T,class P> void quicksort(T s[],int iL,int iR,int d){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T || d<=0) return;
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a,d-1);quicksort<T,P>(s,a+1,iR,d-1);}
template <class T,class P> void quicksort(T s[],int n){
        quicksort<T,P>(s,0,n);
        // take advantage of the best-case performance of insertion sort by a
        // single final call to insertion sort to process the entire array
        insertionsort<T,P>(s,n);}
// END quick sort: divide & conquer
```

Sorting

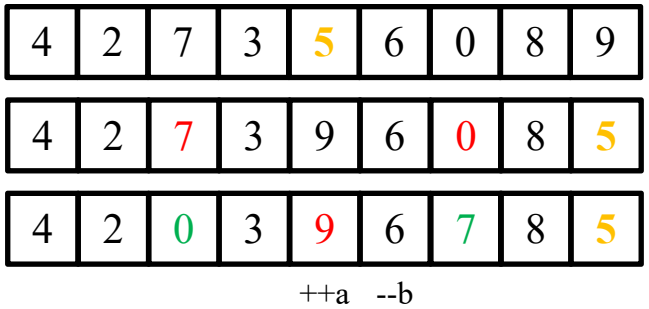
- **Quicksort - divide & conquer**
  - *pivot partition in partially-sorted form (PSF)*
  - partition trick




```
// quick sort: divide & conquer
#define QUICKSORT_SMALL_T 3
#define M_OF_3 0
template <class T> inline int findpivot(T s[],int iL,int iR){return (iL+iR-1)/2;}
template <class T,class P> inline int findpivot(T s[],int iL,int iR){
        int iM=(iL+iR-1)/2;if(P::p(s[iL],s[iM])==P::p(s[iM],s[iR-1])) return iM;
        else if(P::p(s[iM],s[iL])==P::p(s[iL],s[iR-1])) return iL;else return iR-1;}
template <class T,class P> void quicksort(T s[],int iL,int iR){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T) return; // do nothing for small sub-sequences
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a);quicksort<T,P>(s,a+1,iR);}
template <class T,class P> void quicksort(T s[],int iL,int iR,int d){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T || d<=0) return;
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a,d-1);quicksort<T,P>(s,a+1,iR,d-1);}
template <class T,class P> void quicksort(T s[],int n){
        quicksort<T,P>(s,0,n);
        // take advantage of the best-case performance of insertion sort by a
        // single final call to insertion sort to process the entire array
        insertionsort<T,P>(s,n);}
// END quick sort: divide & conquer
```

Sorting

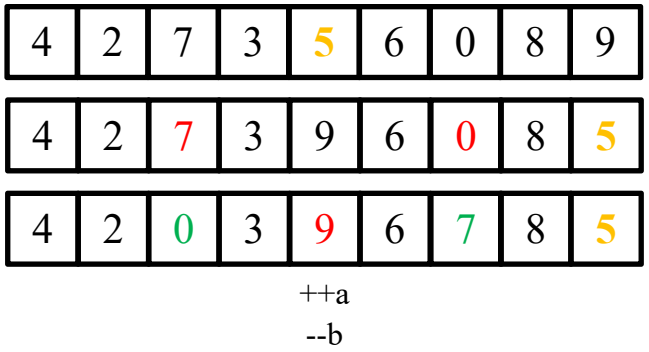
- **Quicksort - divide & conquer**
  - *pivot partition in partially-sorted form (PSF)*
  - partition trick




```
// quick sort: divide & conquer
#define QUICKSORT_SMALL_T 3
#define M_OF_3 0
template <class T> inline int findpivot(T s[],int iL,int iR){return (iL+iR-1)/2;}
template <class T,class P> inline int findpivot(T s[],int iL,int iR){
        int iM=(iL+iR-1)/2;if(P::p(s[iL],s[iM])==P::p(s[iM],s[iR-1])) return iM;
        else if(P::p(s[iM],s[iL])==P::p(s[iL],s[iR-1])) return iL;else return iR-1;}
template <class T,class P> void quicksort(T s[],int iL,int iR){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T) return; // do nothing for small sub-sequences
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a);quicksort<T,P>(s,a+1,iR);}
template <class T,class P> void quicksort(T s[],int iL,int iR,int d){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T || d<=0) return;
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a,d-1);quicksort<T,P>(s,a+1,iR,d-1);}
template <class T,class P> void quicksort(T s[],int n){
        quicksort<T,P>(s,0,n);
        // take advantage of the best-case performance of insertion sort by a
        // single final call to insertion sort to process the entire array
        insertionsort<T,P>(s,n);}
// END quick sort: divide & conquer
```

Sorting

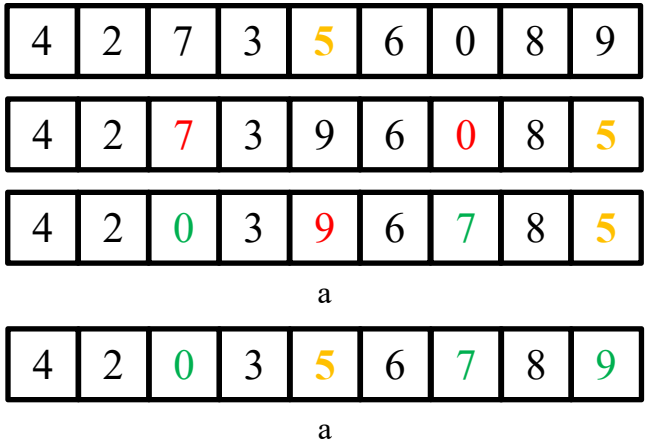
- **Quicksort - divide & conquer**
  - *pivot partition in partially-sorted form (PSF)*
  - partition trick




```
// quick sort: divide & conquer
#define QUICKSORT_SMALL_T 3
#define M_OF_3 0
template <class T> inline int findpivot(T s[],int iL,int iR){return (iL+iR-1)/2;}
template <class T,class P> inline int findpivot(T s[],int iL,int iR){
        int iM=(iL+iR-1)/2;if(P::p(s[iL],s[iM])==P::p(s[iM],s[iR-1])) return iM;
        else if(P::p(s[iM],s[iL])==P::p(s[iL],s[iR-1])) return iL;else return iR-1;}
template <class T,class P> void quicksort(T s[],int iL,int iR){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T) return; // do nothing for small sub-sequences
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a);quicksort<T,P>(s,a+1,iR);}
template <class T,class P> void quicksort(T s[],int iL,int iR,int d){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T || d<=0) return;
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a,d-1);quicksort<T,P>(s,a+1,iR,d-1);}
template <class T,class P> void quicksort(T s[],int n){
        quicksort<T,P>(s,0,n);
        // take advantage of the best-case performance of insertion sort by a
        // single final call to insertion sort to process the entire array
        insertionsort<T,P>(s,n);}
// END quick sort: divide & conquer
```

Sorting

- **Quicksort - divide & conquer**
  - *pivot partition in partially-sorted form (PSF)*
  - partition trick




```
// quick sort: divide & conquer
#define QUICKSORT_SMALL_T 3
#define M_OF_3 0
template <class T> inline int findpivot(T s[],int iL,int iR){return (iL+iR-1)/2;}
template <class T,class P> inline int findpivot(T s[],int iL,int iR){
        int iM=(iL+iR-1)/2;if(P::p(s[iL],s[iM])==P::p(s[iM],s[iR-1])) return iM;
        else if(P::p(s[iM],s[iL])==P::p(s[iL],s[iR-1])) return iL;else return iR-1;}
template <class T,class P> void quicksort(T s[],int iL,int iR){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T) return; // do nothing for small sub-sequences
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a);quicksort<T,P>(s,a+1,iR);}
template <class T,class P> void quicksort(T s[],int iL,int iR,int d){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T || d<=0) return;
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a,d-1);quicksort<T,P>(s,a+1,iR,d-1);}
template <class T,class P> void quicksort(T s[],int n){
        quicksort<T,P>(s,0,n);
        // take advantage of the best-case performance of insertion sort by a
        // single final call to insertion sort to process the entire array
        insertionsort<T,P>(s,n);}
// END quick sort: divide & conquer
```

Sorting

- **Quicksort - divide & conquer**
  - *pivot partition in partially-sorted form (PSF)*
  - partition trick
  - do nothing for small

```
// quick sort: divide & conquer
#define QUICKSORT_SMALL_T 3
#define M_OF_3 0
template <class T> inline int findpivot(T s[],int iL,int iR){return (iL+iR-1)/2;}
template <class T,class P> inline int findpivot(T s[],int iL,int iR){
        int iM=(iL+iR-1)/2;if(P::p(s[iL],s[iM])==P::p(s[iM],s[iR-1])) return iM;
        else if(P::p(s[iM],s[iL])==P::p(s[iL],s[iR-1])) return iL;else return iR-1;}
template <class T,class P> void quicksort(T s[],int iL,int iR){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T) return; // do nothing for small sub-sequences
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a);quicksort<T,P>(s,a+1,iR);}
template <class T,class P> void quicksort(T s[],int iL,int iR,int d){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T || d<=0) return;
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a,d-1);quicksort<T,P>(s,a+1,iR,d-1);}
template <class T,class P> void quicksort(T s[],int n){
        quicksort<T,P>(s,0,n);
        // take advantage of the best-case performance of insertion sort by a
        // single final call to insertion sort to process the entire array
        insertionsort<T,P>(s,n);}
// END quick sort: divide & conquer
```

Sorting

- **Quicksort - divide & conquer**
  - *pivot partition in partially-sorted form (PSF)*
  - partition trick
  - do nothing for small
  - final insertion sort

```
// quick sort: divide & conquer
#define QUICKSORT_SMALL_T 3
#define M_OF_3 0
template <class T> inline int findpivot(T s[],int iL,int iR){return (iL+iR-1)/2;}
template <class T,class P> inline int findpivot(T s[],int iL,int iR){
        int iM=(iL+iR-1)/2;if(P::p(s[iL],s[iM])==P::p(s[iM],s[iR-1])) return iM;
        else if(P::p(s[iM],s[iL])==P::p(s[iL],s[iR-1])) return iL;else return iR-1;}
template <class T,class P> void quicksort(T s[],int iL,int iR){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T) return; // do nothing for small sub-sequences
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a);quicksort<T,P>(s,a+1,iR);}
template <class T,class P> void quicksort(T s[],int iL,int iR,int d){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T || d<=0) return;
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a,d-1);quicksort<T,P>(s,a+1,iR,d-1);}
template <class T,class P> void quicksort(T s[],int n){
        quicksort<T,P>(s,0,n);
        // take advantage of the best-case performance of insertion sort by a
        // single final call to insertion sort to process the entire array
        insertionsort<T,P>(s,n);}
// END quick sort: divide & conquer
```

Sorting

- **Quicksort - divide & conquer**
  - *pivot partition in partially-sorted form (PSF)*
  - partition trick
  - do nothing for small
  - final insertion sort

```
// quick sort: divide & conquer
#define QUICKSORT_SMALL_T 3
#define M_OF_3 0
template <class T> inline int findpivot(T s[],int iL,int iR){return (iL+iR-1)/2;}
template <class T,class P> inline int findpivot(T s[],int iL,int iR){
        int iM=(iL+iR-1)/2;if(P::p(s[iL],s[iM])==P::p(s[iM],s[iR-1])) return iM;
        else if(P::p(s[iM],s[iL])==P::p(s[iL],s[iR-1])) return iL;else return iR-1;}
template <class T,class P> void quicksort(T s[],int iL,int iR){ // [iL,iR)
        if((iR-iL)<=QUICKSORT_SMALL_T) return; // do nothing for small sub-sequences
        int iP;if(M_OF_3) iP=findpivot<T,P>(s,iL,iR);else iP=findpivot<T>(s,iL,iR);
        swap(s,iP,iR-1);iP=iR-1;int a=iL-1,b=iP;
        do{while((a<b)&&P::p(s[++a],s[iP]));while((a<b)&&P::p(s[iP],s[--b]));
                swap(s,a,b);}while(a<b);
        swap(s,a,iP);quicksort<T,P>(s,iL,a);quicksort<T,P>(s,a+1,iR);}
```

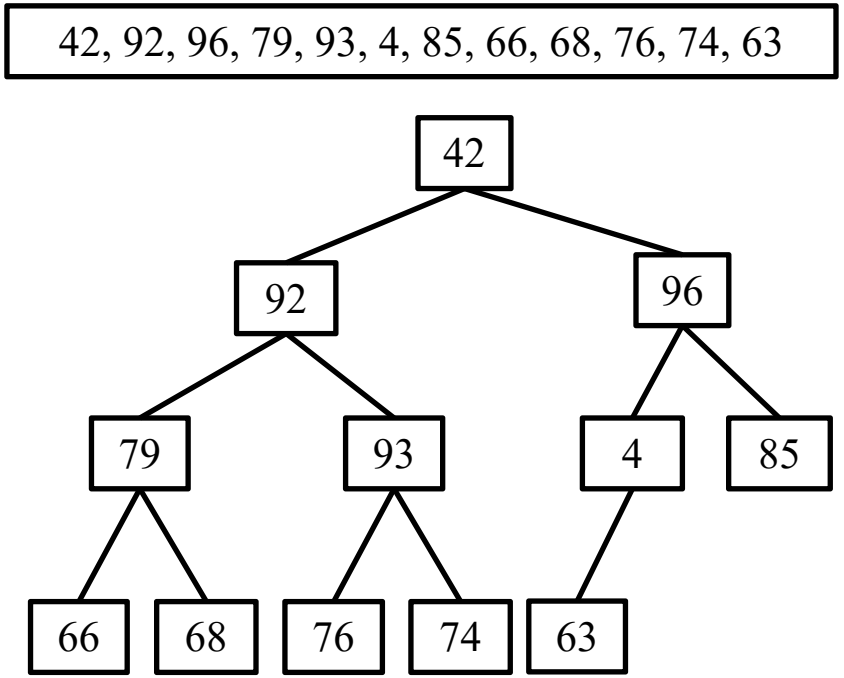
```
DEMO : quick sort =>
unsorted sequence: 42 92 96 79 93 4 85 66 68 76 74 63 39 17 71 3
find pivot (median of three) => 42
quick sort: 3 17 39 4 42 79 85 66 68 76 74 63 96 92 71 93
quick sort: 3 4 39 17 42 71 63 66 68 76 74 79 96 92 93 85
quick sort: 3 4 39 17 42 68 63 66 71 76 74 79 85 92 93 96
quick sort (with final insertionsort): 3 4 17 39 42 63 66 68 71 74 76 79 85 92 93 96
unsorted sequence: machine intelligence system automation program technique computer data
quick sort: automation computer data intelligence machine program system technique
```

```
cout<<"DEMO : quick sort =>\n";
cp<int>(iA,iTab,ni);cp<cptr>(cA,cTab,nc);
cout<<"unsorted sequence: ";show<int>(iA,ni);
int iP;if(M_OF_3) iP=findpivot<int,IntPriorMin>(iA,0,ni);else iP=findpivot<int>(iA,0,ni);
if(M_OF_3) cout<<"find pivot (median of three) => "<<iA[iP]<<endl;
else cout<<"find pivot (simply middle) => "<<iA[iP]<<endl;
cout<<"quick sort: ";quicksort<int,IntPriorMin>(iA,0,ni,1);show<int>(iA,ni);
cout<<"quick sort: ";cp<int>(iA,iTab,ni);quicksort<int,IntPriorMin>(iA,0,ni,2);show<int>(iA,ni);
cout<<"quick sort: ";cp<int>(iA,iTab,ni);quicksort<int,IntPriorMin>(iA,0,ni);show<int>(iA,ni);
cout<<"quick sort (with final insertionsort): ";quicksort<int,IntPriorMin>(iA,ni);show<int>(iA,ni);
cout<<"unsorted sequence: ";show<cptr>(cA,nc);
cout<<"quick sort: ";quicksort<cptr,CharsPriorMin>(cA,nc);show<cptr>(cA,nc);
```

```
DEMO : quick sort =>
unsorted sequence: 42 92 96 79 93 4 85 66 68 76 74 63 39 17 71 3
find pivot (simply middle) => 66
quick sort: 42 17 39 63 3 4 66 93 68 76 74 79 96 92 71 85
quick sort: 3 17 4 39 42 63 66 71 68 76 74 79 96 92 93 85
quick sort: 3 17 4 39 42 63 66 68 74 76 71 79 85 92 93 96
quick sort (with final insertionsort): 3 4 17 39 42 63 66 68 71 74 76 79 85 92 93 96
unsorted sequence: machine intelligence system automation program technique computer da
quick sort: automation computer data intelligence machine program system technique
```

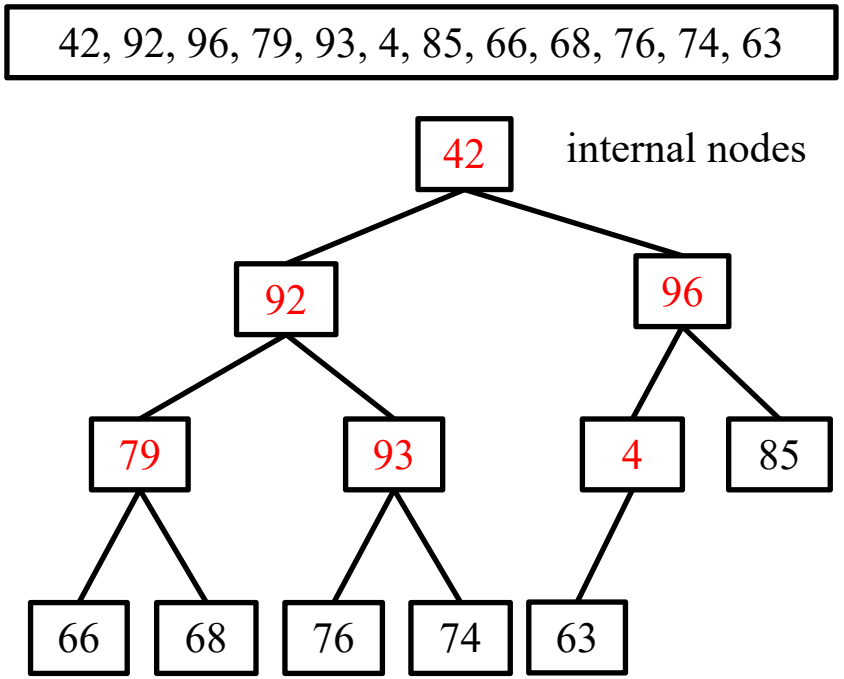
Sorting

- **Heap sort**
  - *batch initialization*
  - *iterative removeroot*



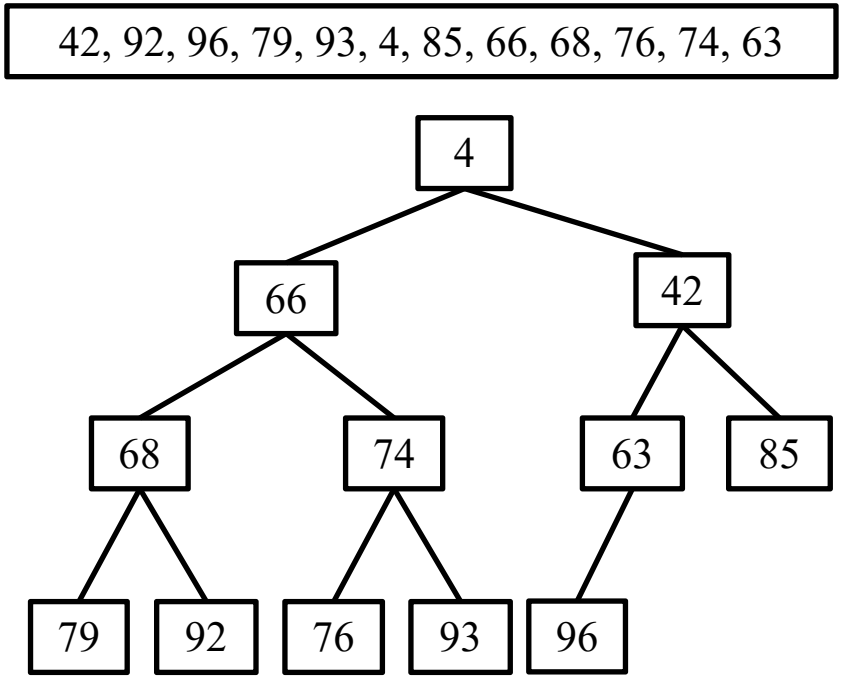
Sorting

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
  - *iterative removeroot*



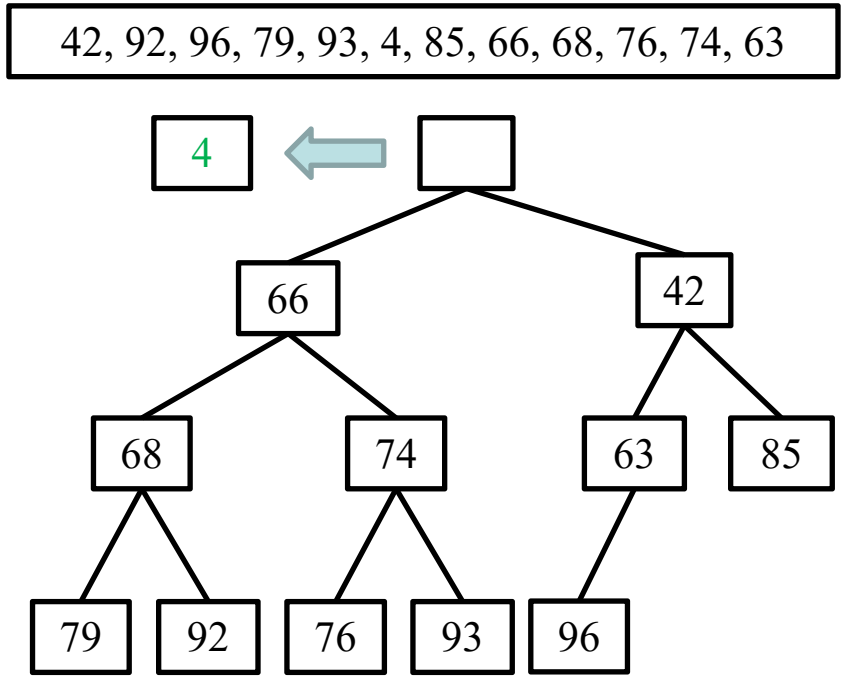
Sorting

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*



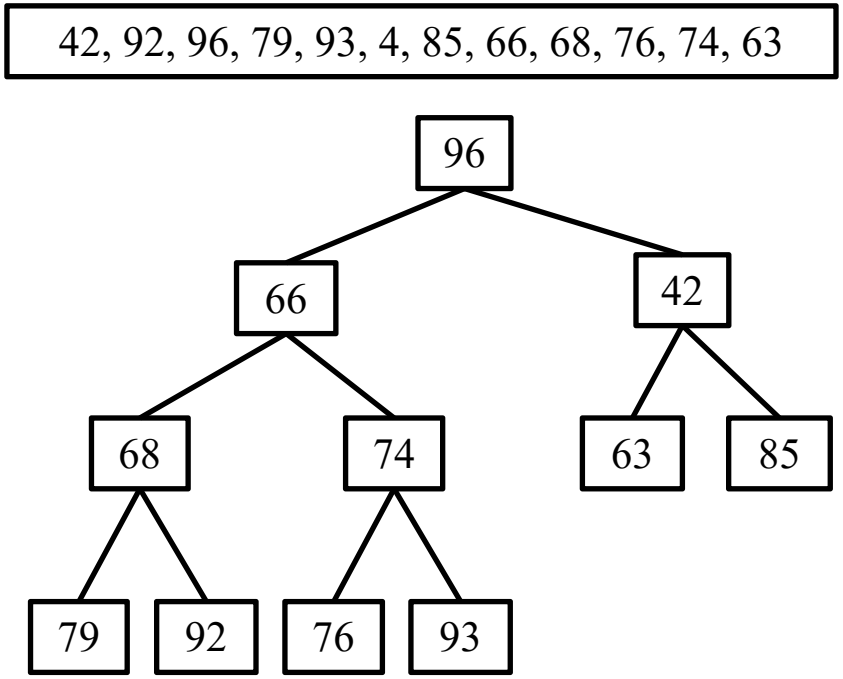
Sorting

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*



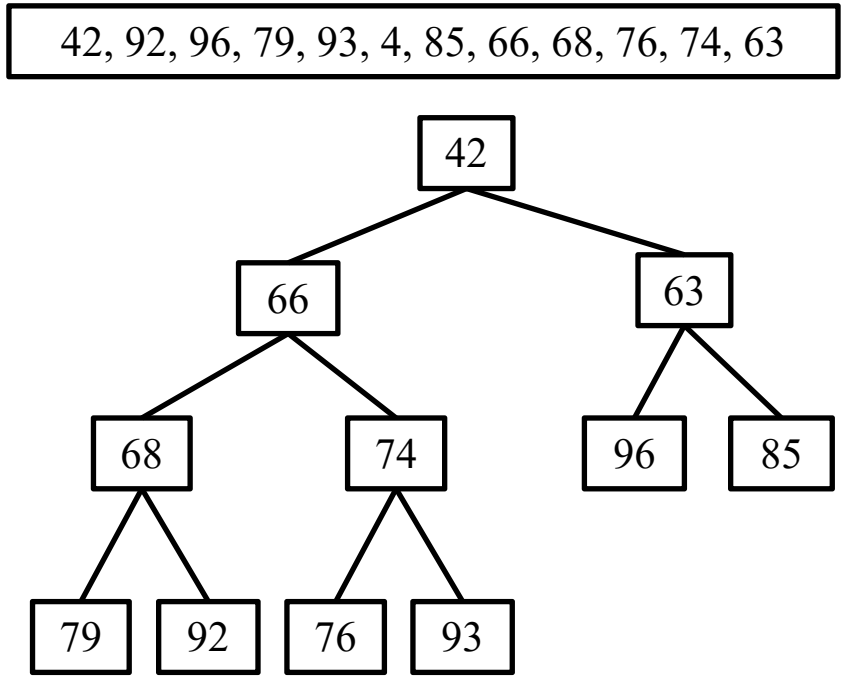
Sorting

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*



Sorting

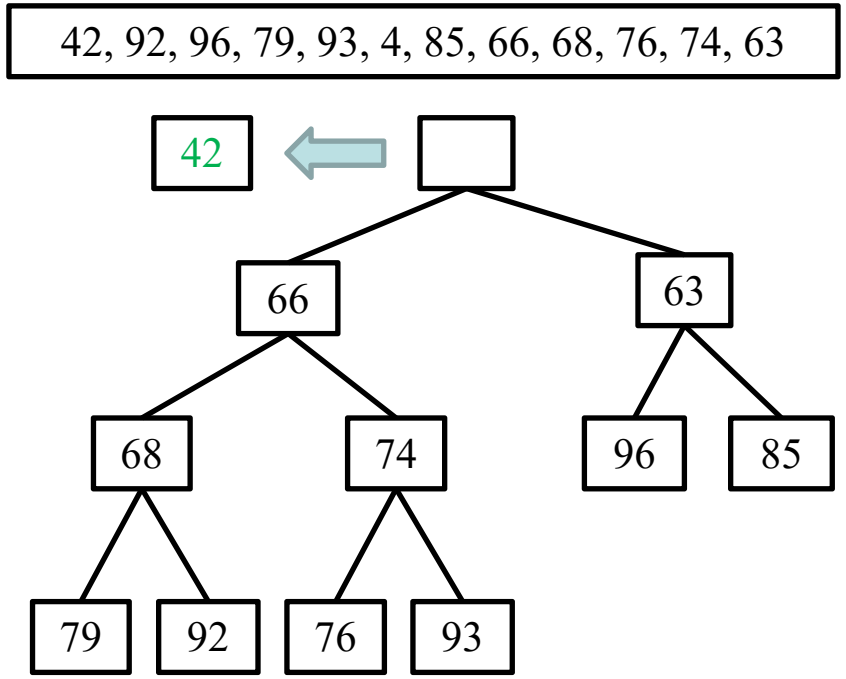
- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*



4
42

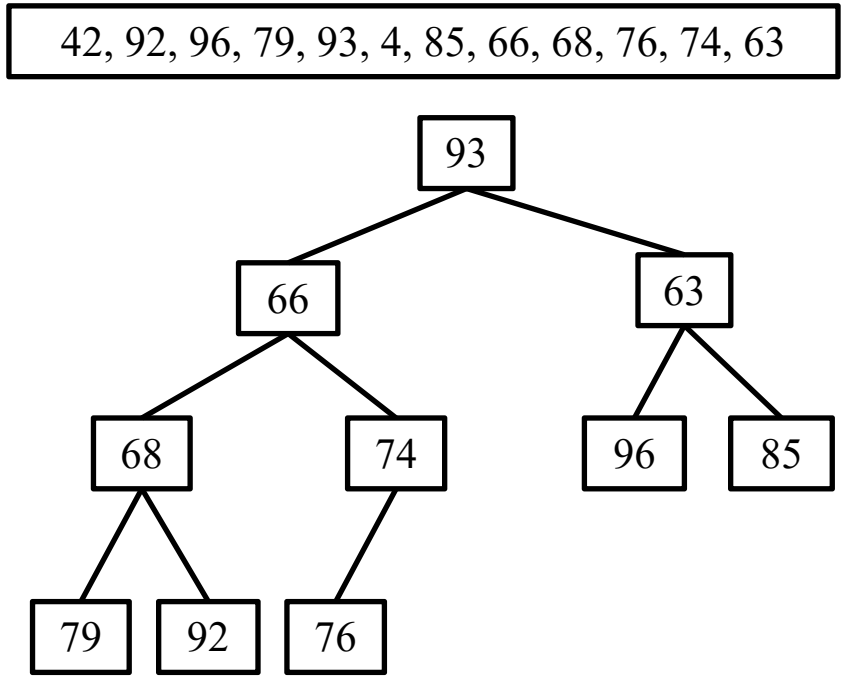
Sorting

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*



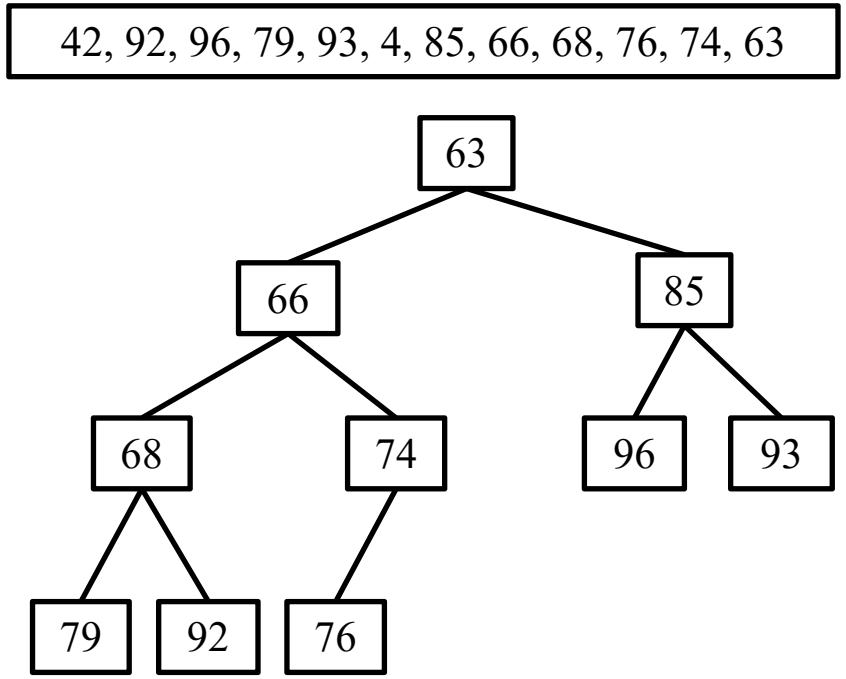
Sorting

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*



Sorting

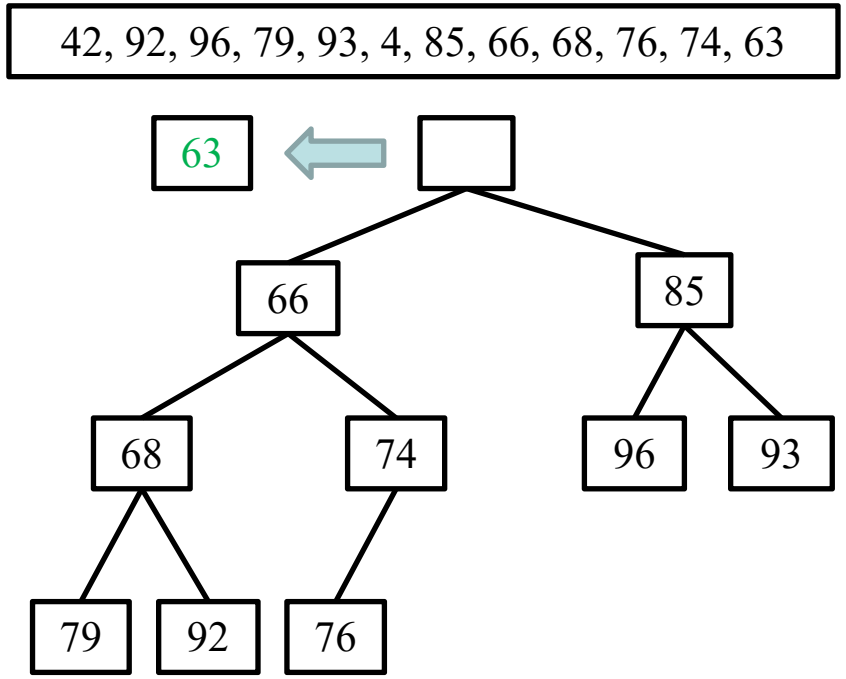
- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*



4
42
63

Sorting

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*

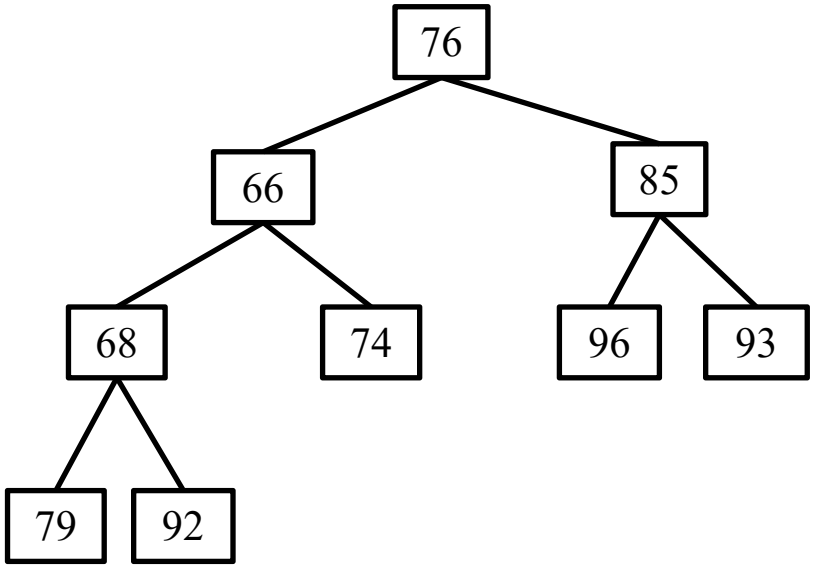


4
42
63

Sorting

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*

42, 92, 96, 79, 93, 4, 85, 66, 68, 76, 74, 63

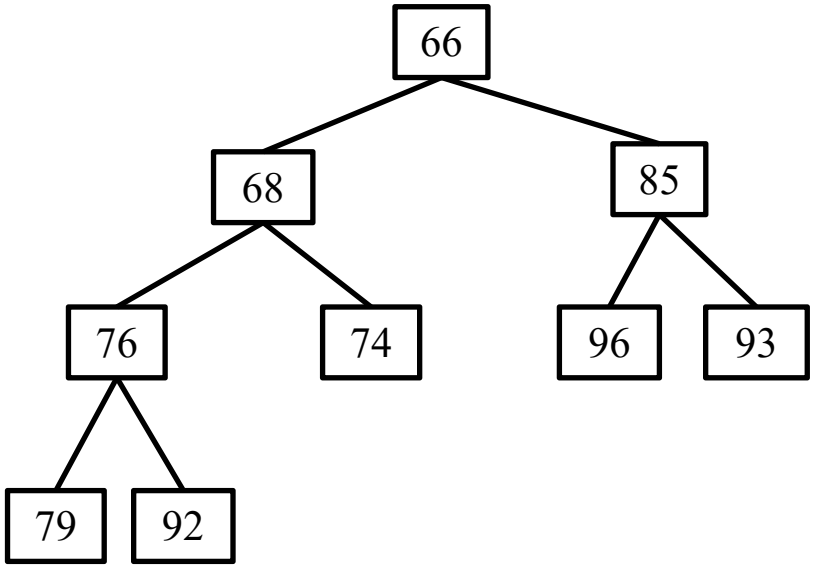


4
42
63

Sorting

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*

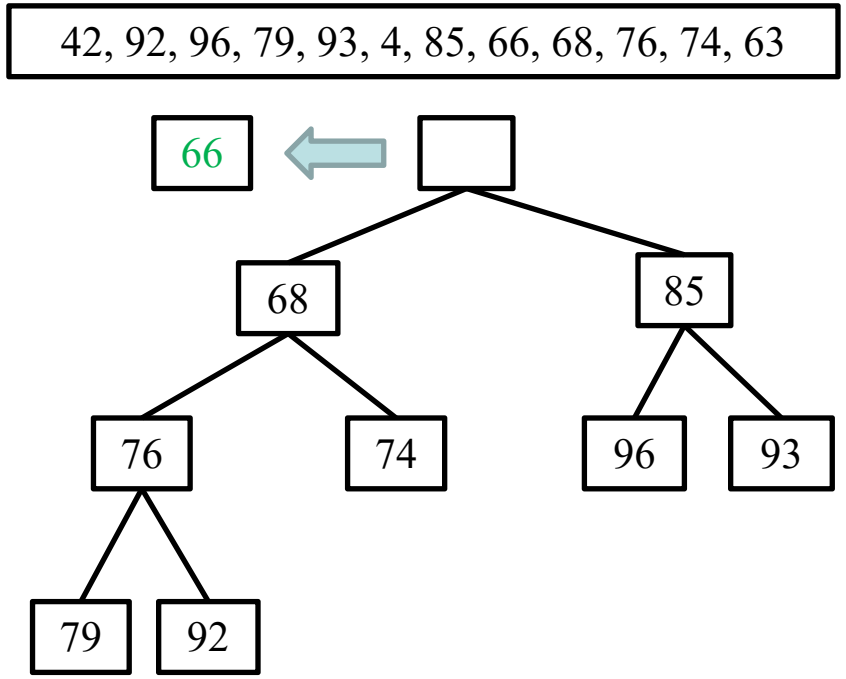
42, 92, 96, 79, 93, 4, 85, 66, 68, 76, 74, 63



4
42
63
66

Sorting

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*

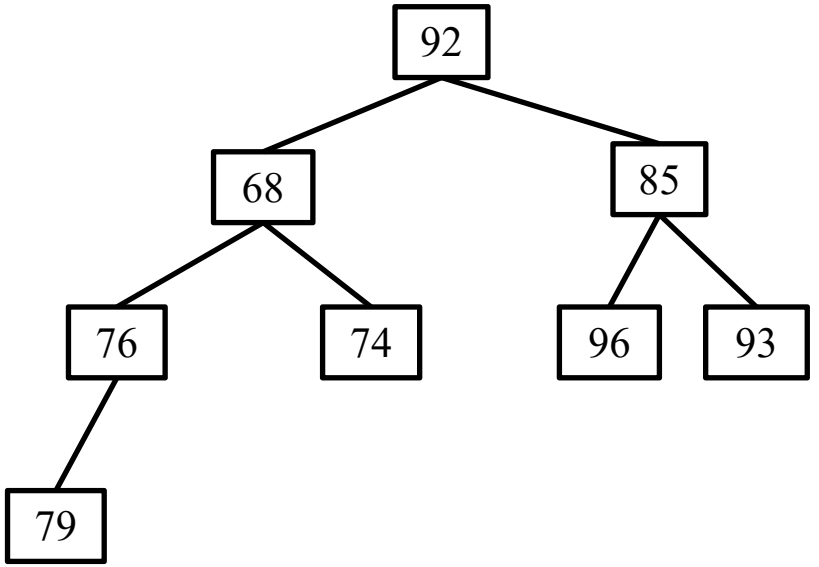


4
42
63
66

Sorting

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*

42, 92, 96, 79, 93, 4, 85, 66, 68, 76, 74, 63

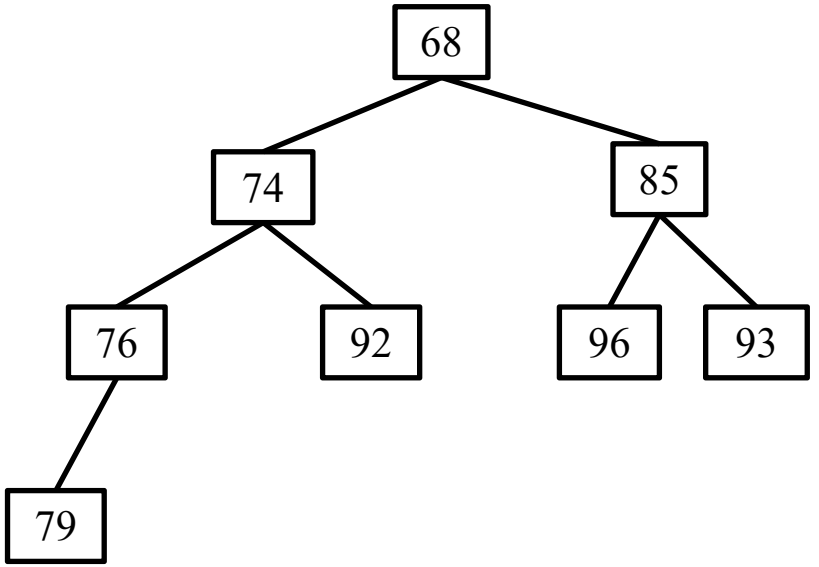


4
42
63
66

Sorting

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*

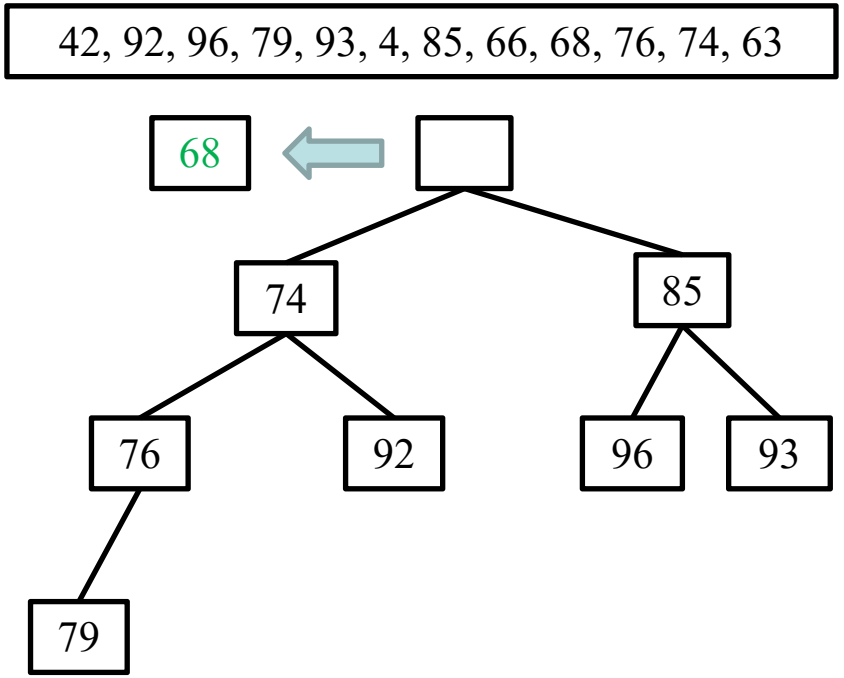
42, 92, 96, 79, 93, 4, 85, 66, 68, 76, 74, 63



4
42
63
66
68

Sorting

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*

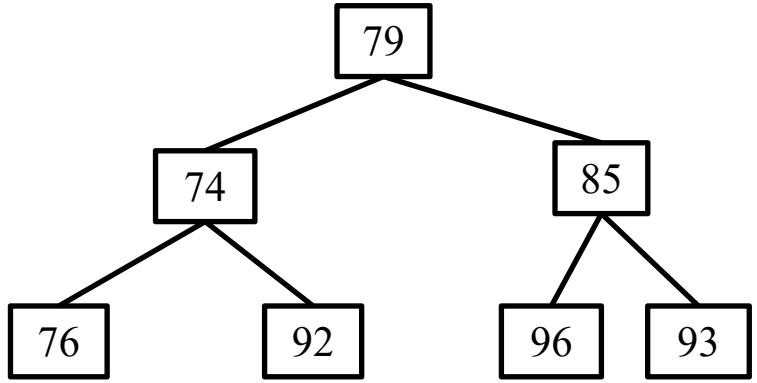


4
42
63
66
68

Sorting

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*

42, 92, 96, 79, 93, 4, 85, 66, 68, 76, 74, 63

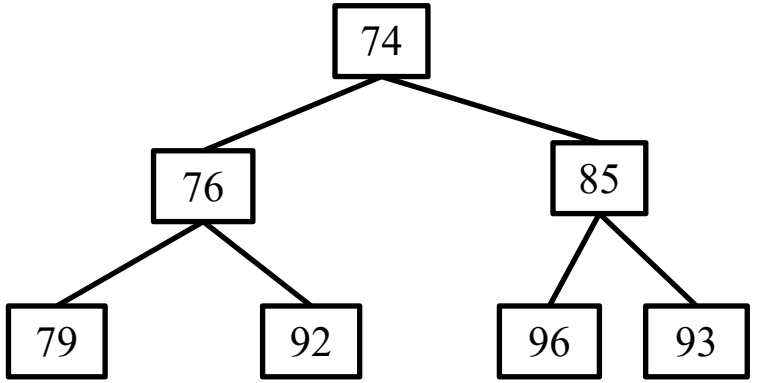


4
42
63
66
68

Sorting

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*

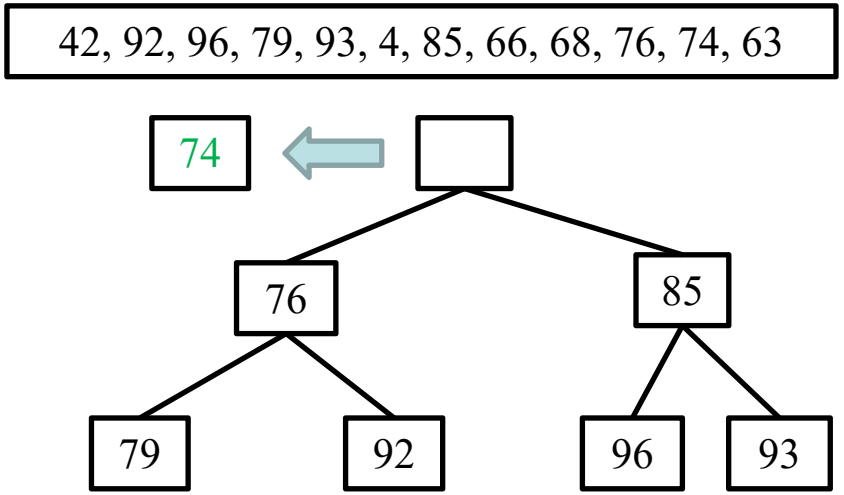
42, 92, 96, 79, 93, 4, 85, 66, 68, 76, 74, 63



4
42
63
66
68
74

Sorting

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*

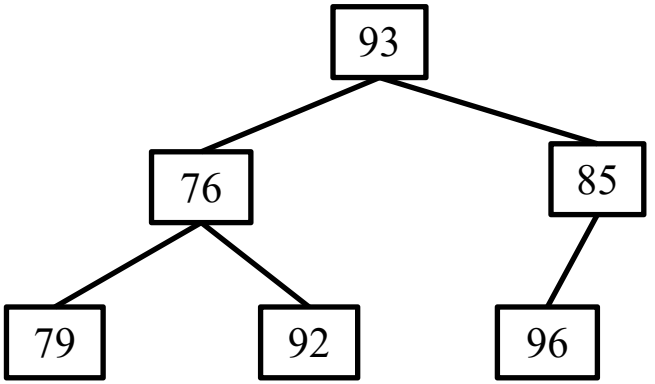


4
42
63
66
68
74

Sorting

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*

42, 92, 96, 79, 93, 4, 85, 66, 68, 76, 74, 63

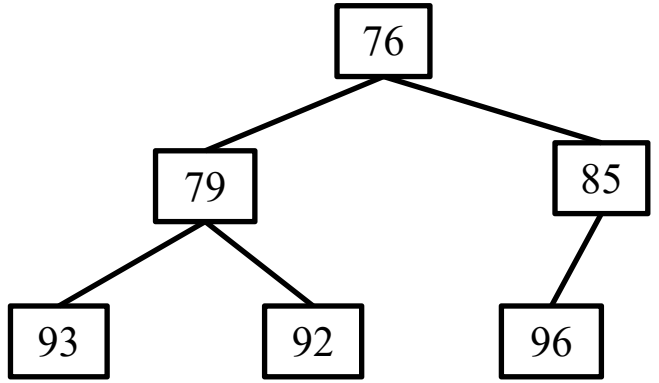


4
42
63
66
68
74

Sorting

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*

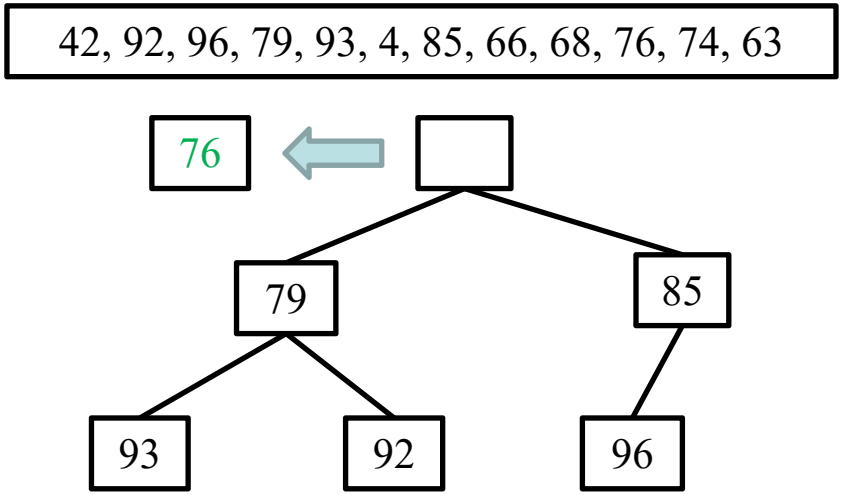
42, 92, 96, 79, 93, 4, 85, 66, 68, 76, 74, 63



Sorting

4
42
63
66
68
74
76

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*

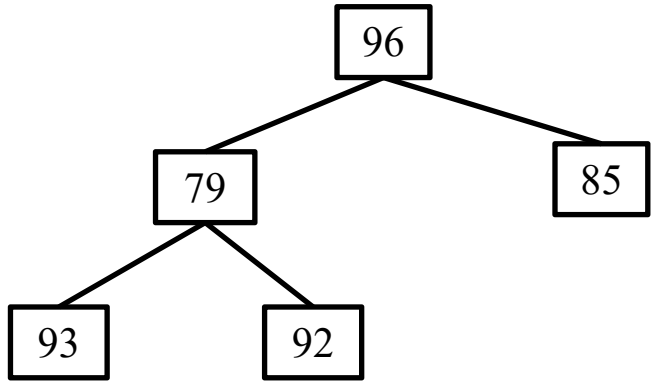


Sorting

4
42
63
66
68
74
76

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*

42, 92, 96, 79, 93, 4, 85, 66, 68, 76, 74, 63

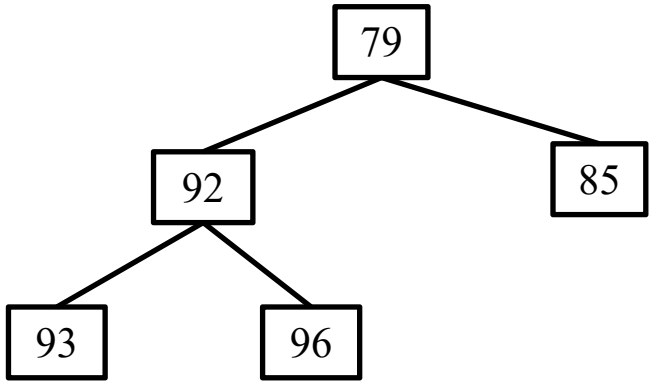


Sorting

4
42
63
66
68
74
76

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*

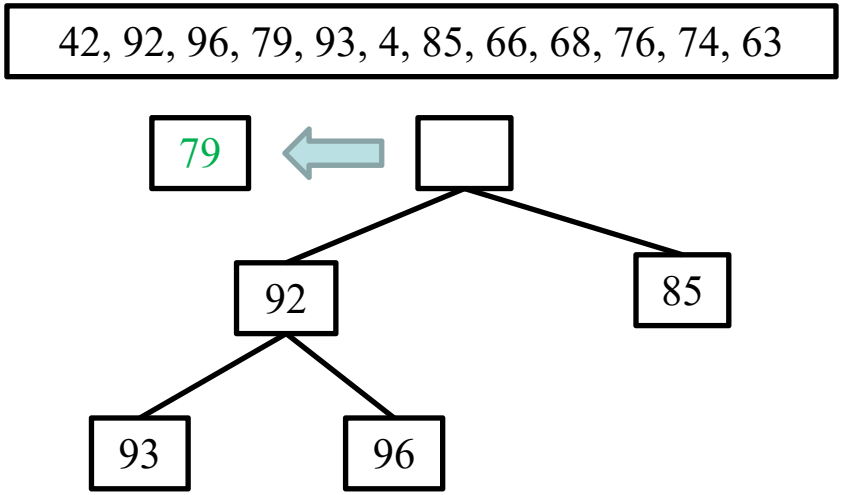
42, 92, 96, 79, 93, 4, 85, 66, 68, 76, 74, 63



Sorting

4
42
63
66
68
74
76
79

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*

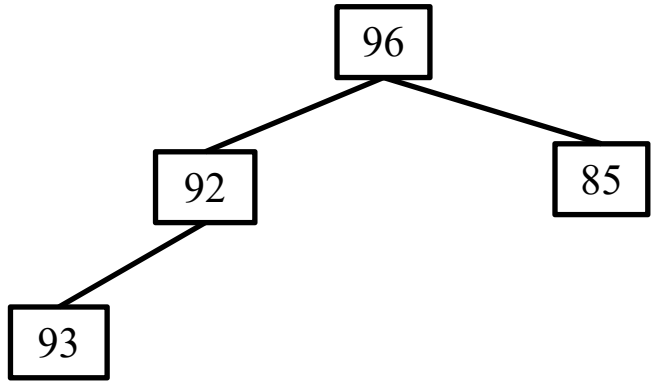


Sorting

4
42
63
66
68
74
76
79

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*

42, 92, 96, 79, 93, 4, 85, 66, 68, 76, 74, 63

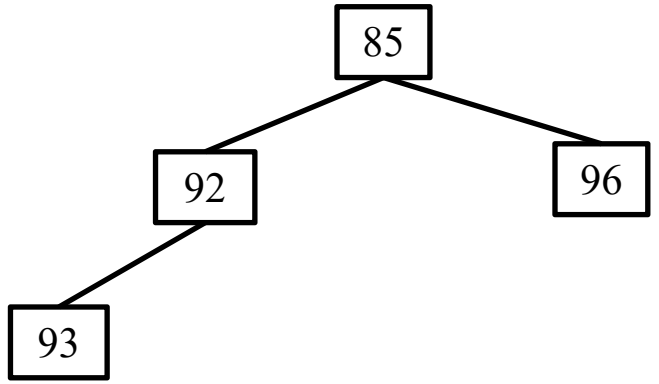


Sorting

4
42
63
66
68
74
76
79

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*

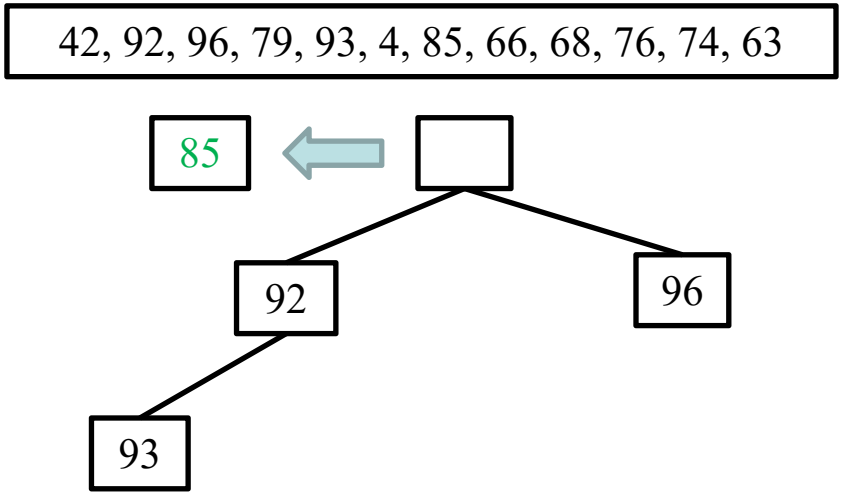
42, 92, 96, 79, 93, 4, 85, 66, 68, 76, 74, 63



Sorting

4
42
63
66
68
74
76
79
85

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*

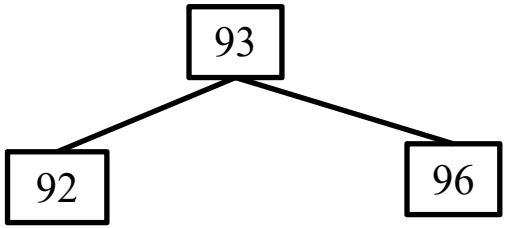


Sorting

4
42
63
66
68
74
76
79
85

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*

42, 92, 96, 79, 93, 4, 85, 66, 68, 76, 74, 63

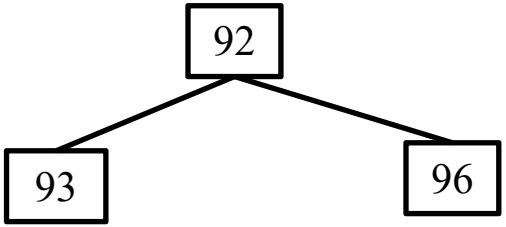


4
42
63
66
68
74
76
79
85

Sorting

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*

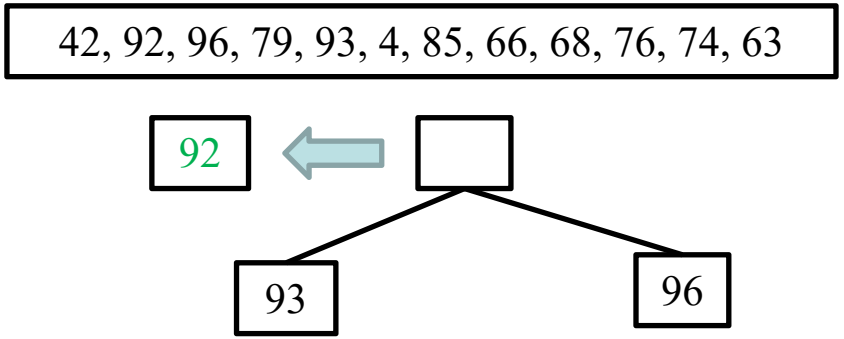
42, 92, 96, 79, 93, 4, 85, 66, 68, 76, 74, 63



Sorting

4
42
63
66
68
74
76
79
85
92

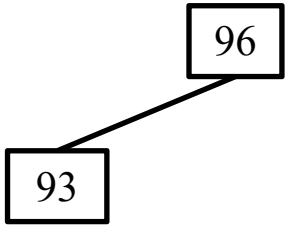
- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*



Sorting

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*

42, 92, 96, 79, 93, 4, 85, 66, 68, 76, 74, 63

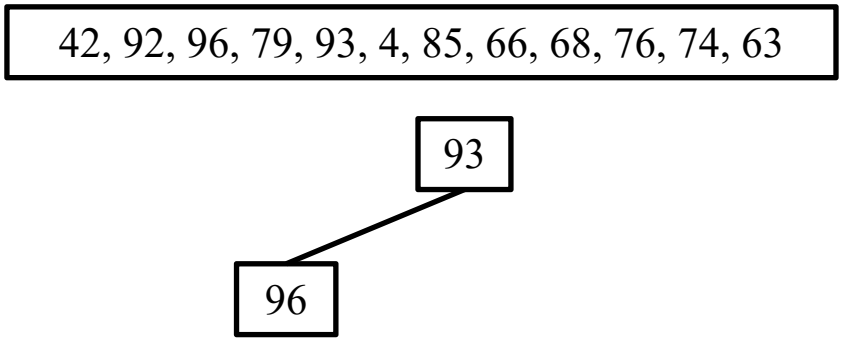


4
42
63
66
68
74
76
79
85
92

Sorting

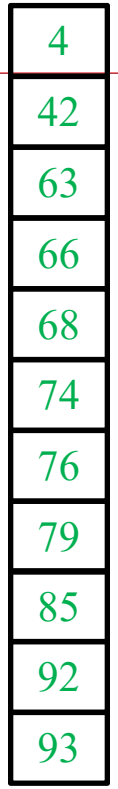
4
42
63
66
68
74
76
79
85
92

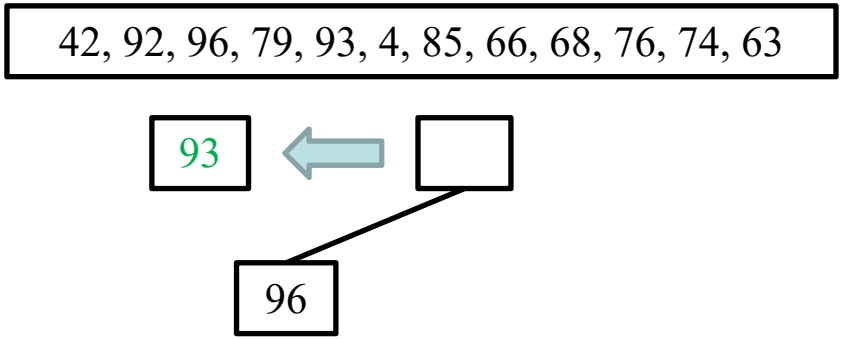
- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*



Sorting

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*





Sorting

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*

42, 92, 96, 79, 93, 4, 85, 66, 68, 76, 74, 63

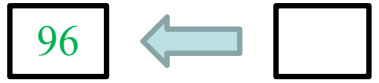
96

4
42
63
66
68
74
76
79
85
92
93

Sorting

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*

42, 92, 96, 79, 93, 4, 85, 66, 68, 76, 74, 63

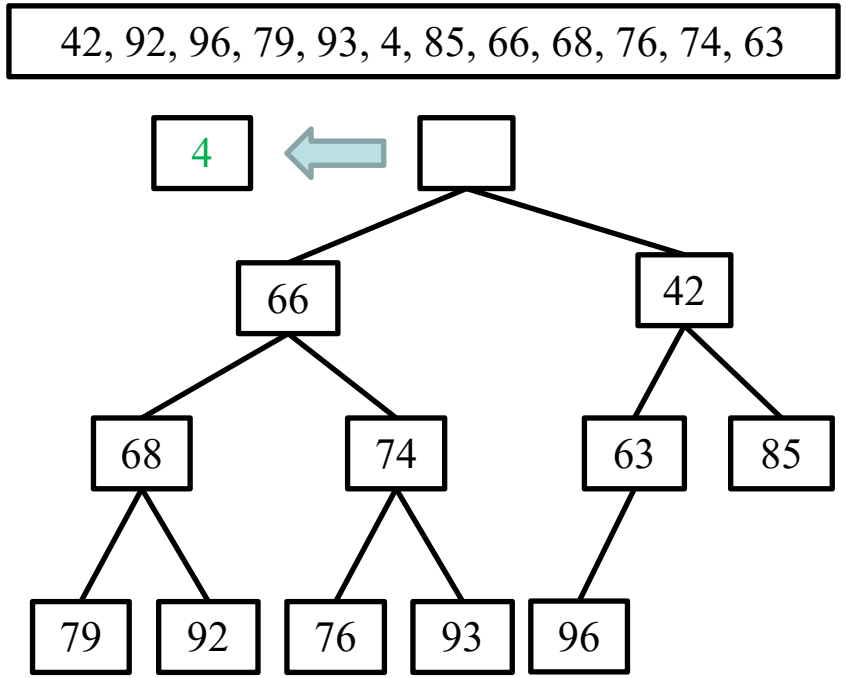


4
42
63
66
68
74
76
79
85
92
93
96

Sorting

4
42
63
66
68
74
76
79
85
92
93
96

- **Heap sort**
  - *batch initialization*
    - backward iterated siftdown
    - complexity: O(n)
  - *iterative removeroot*
    - each removal involves a siftdown
    - siftdown complexity: O(*log* n)
    - totally n removals
    - total complexity: O(n *log* n)



Sorting

- **Heap sort**
  - *batch initialization*
  - *iterative removeroot*

```
// heap sort
template <class T,class P> void heapsort(T s[],int n){
        Heap<T,P> h(n,s,n);
        for(int i=0;i<n;i++) s[i]=h.removeroot();}
// END heap sort
```

```
cout<<"DEMO : heap sort =>\n";
cp<int>(iA,iTab,ni);cp<cptr>(cA,cTab,nc);
cout<<"unsorted sequence: ";show<int>(iA,ni);
cout<<"heap sort: ";heapsort<int,IntPriorMin>(iA,ni);show<int>(iA,ni);
cout<<"unsorted sequence: ";show<cptr>(cA,nc);
cout<<"heap sort: ";heapsort<cptr,CharsPriorMin>(cA,nc);show<cptr>(cA,nc);
cp<cptr>(cA,cTab,nc);Heap<cptr,CharsPriorMin> h(nc,cA,nc);h.S();
for(int i=0;i<nc;i++){cA[i]=h.removeroot();cout<<"heap sort: ";show<cptr>(cA,i+1);h.S();}
```

```
DEMO : heap sort =>
unsorted sequence: 42 92 96 79 93 4 85 66 68 76 74 63 39 17 71 3
heap sort: 3 4 17 39 42 63 66 68 71 74 76 79 85 92 93 96
unsorted sequence: machine intelligence system automation program technique computer data
heap sort: automation computer data intelligence machine program system technique
```

So

- **Heap sort**
  - *batch initialization*
  - *iterative removeroot*

```
// heap sort
template <class T,class P> void heapsort(T s[],int n){
        Heap<T,P> h(n,s,n);
        for(int i=0;i<n;i++) s[i]=h.removeroot();}
// END heap sort
```

```
cout<<"DEMO : heap sort =>\n";
cp<int>(iA,iTab,ni);cp<cptr>(cA,cTab,nc);
cout<<"unsorted sequence: ";show<int>(iA,ni);
cout<<"heap sort: ";heapsort<int,IntPriorMin>(iA,ni);show<int>(iA,ni);
cout<<"unsorted sequence: ";show<cptr>(cA,nc);
cout<<"heap sort: ";heapsort<cptr,CharsPriorMin>(cA,nc);show<cptr>(cA,nc);
cp<cptr>(cA,cTab,nc);Heap<cptr,CharsPriorMin> h(nc,cA,nc);h.S();
for(int i=0;i<nc;i++){cA[i]=h.removeroot();cout<<"heap sort: ";show<cptr>(cA,i+1);h.S();}
```

```
unsorted sequence: machine intelligence system automation program technique computer data
heap sort: automation computer data intelligence machine program system technique
                    7:machine
              3:intelligence
        1:data
              4:program
0:automation
              5:technique
        2:computer
              6:system
heap sort: automation
              3:intelligence
        1:data
              4:program
0:computer
              5:technique
        2:machine
              6:system
heap sort: automation computer
              3:system
        1:intelligence
              4:program
0:data
              5:technique
        2:machine
heap sort: automation computer data
              3:system
        1:program
              4:technique
0:intelligence
        2:machine
heap sort: automation computer data intelligence
              3:system
        1:program
0:machine
        2:technique
heap sort: automation computer data intelligence machine
        1:system
0:program
        2:technique
heap sort: automation computer data intelligence machine program
        1:technique
0:system
heap sort: automation computer data intelligence machine program system
0:technique
heap sort: automation computer data intelligence machine program system technique
Heap is empty!
```

Sorting

- **Sorting complexity**
  - insertion sort W:O(n^2) A:O(n^2) B:O(n)
  - bubble sort W:O(n^2) A:O(n^2) B:O(n^2)
  - selection sort W:O(n^2) A:O(n^2) B:O(n^2)
  - shell sort (depending on *shell sequence*) O(n (*log* n)^2) ~ O($n^{1.5}$)
  - mergesort W:O(n *log* n) A:O(n *log* n) B:O(n *log* n)
  - BST sort W:O(n^2) A:O(n *log* n) B:O(n *log* n)
  - quicksort W:O(n^2) A:O(n *log* n) B:O(n *log* n)
  - heap sort W:O(n *log* n) A:O(n *log* n) B:O(n *log* n)

Sorting

- **Sorting complexity**
  - insertion sort W:O(n^2) A:O(n^2) B:O(n)
  - bubble sort W:O(n^2) A:O(n^2) B:O(n^2)
  - selection sort W:O(n^2) A:O(n^2) B:O(n^2)
  - shell sort (depending on *shell sequence*) O(n (*log* n)^2) ~ O($n^{1.5}$)
  - mergesort W:O(n *log* n) A:O(n *log* n) B:O(n *log* n)
  - BST sort W:O(n^2) A:O(n *log* n) B:O(n *log* n)
  - **quicksort** W:O(n^2) A:O(n *log* n) B:O(n *log* n)
  - heap sort W:O(n *log* n) A:O(n *log* n) B:O(n *log* n)
- **Reflection**
  - why is quicksort called “quick”? why is not mergesort or heap sort so called?
  - BST sort seems not so bad, why is BST sort hardly used?

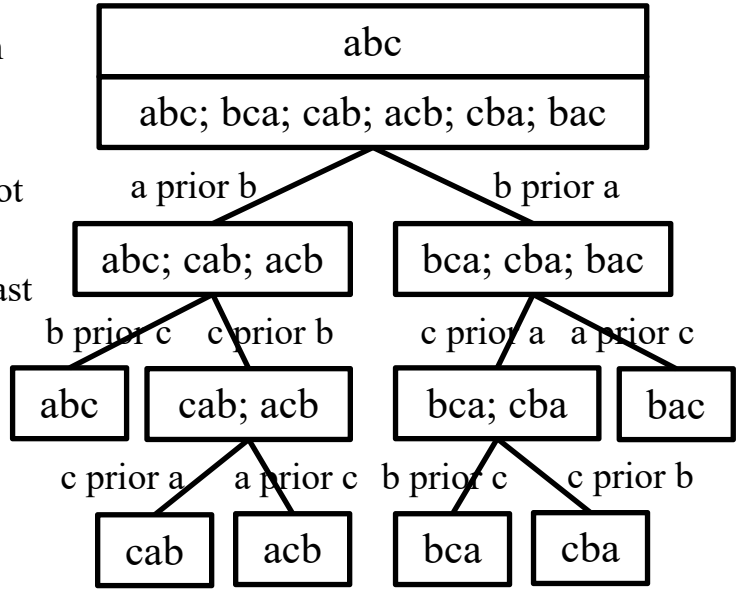
Sorting

- **Sorting complexity**
  - insertion sort W:O(n^2) A:O(n^2) B:O(n)
  - bubble sort W:O(n^2) A:O(n^2) B:O(n^2)
  - selection sort W:O(n^2) A:O(n^2) B:O(n^2)
  - shell sort (depending on *shell sequence*) O(n (*log* n)^2) ~ O($n^{1.5}$)
  - mergesort W:O(n *log* n) A:O(n *log* n) B:O(n *log* n)
  - BST sort W:O(n^2) A:O(n *log* n) B:O(n *log* n)
  - **quicksort** W:O(n^2) A:O(n *log* n) B:O(n *log* n)
  - heap sort W:O(n *log* n) A:O(n *log* n) B:O(n *log* n)
- **Reflection**
  - why is quicksort called “quick”? why is not mergesort or heap sort so called?
  - BST sort seems not so bad, why is BST sort hardly used?

not only complexity level matters, but also does concrete implementation (computation & memory)

Sorting

- **Lower bounds for sorting**
  - worst-case complexity: ≥ O(n *log* n)
  - perspective of *decision tree*
    - each comparison (& potential swap) can be regarded as decision to distinguish between two branches of possible permutations
    - complexity of sorting a permutation equals the depth of decision that leads from the root to the permutation (leaf)
    - the decision tree must have a depth of at least *log*(n!) to accommodate all n! possible permutations
    - *log*(n!) ≈ n *log* n



THANK YOU

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY