アルゴリズム論 10

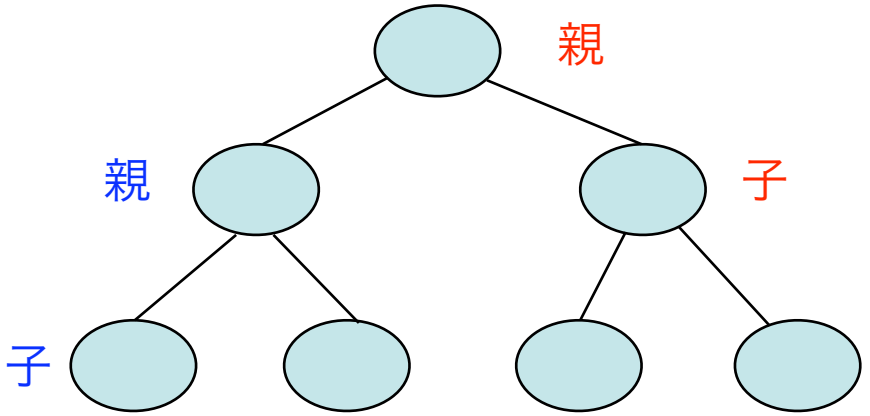
整列処理(ソート)

- バブルソート
- 単純選択ソート
- 挿入法
- クイックソート
- ヒープソート

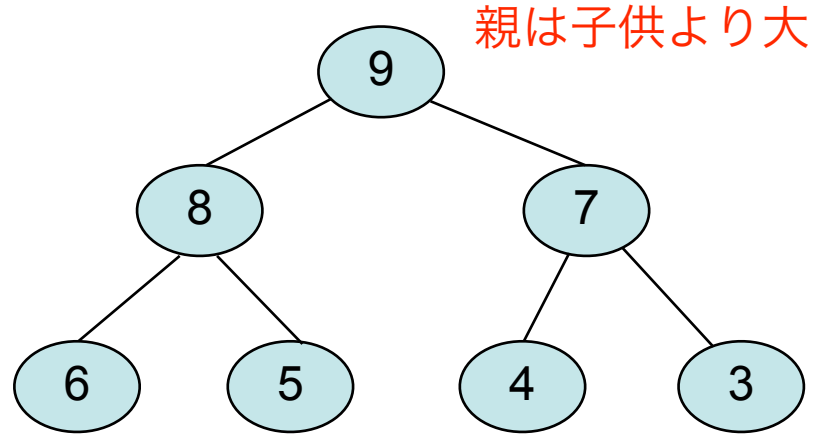
高度な整列処理2(ヒープソート)

ヒープソート

- ヒープ(heap)を使用したソート
- ヒープ
  - 累積、積み重なったもの
  - ヒープソートでは完全2分木を示す
    - 親の値が子の値以上である
- 高速なソートアルゴリズムの一つ



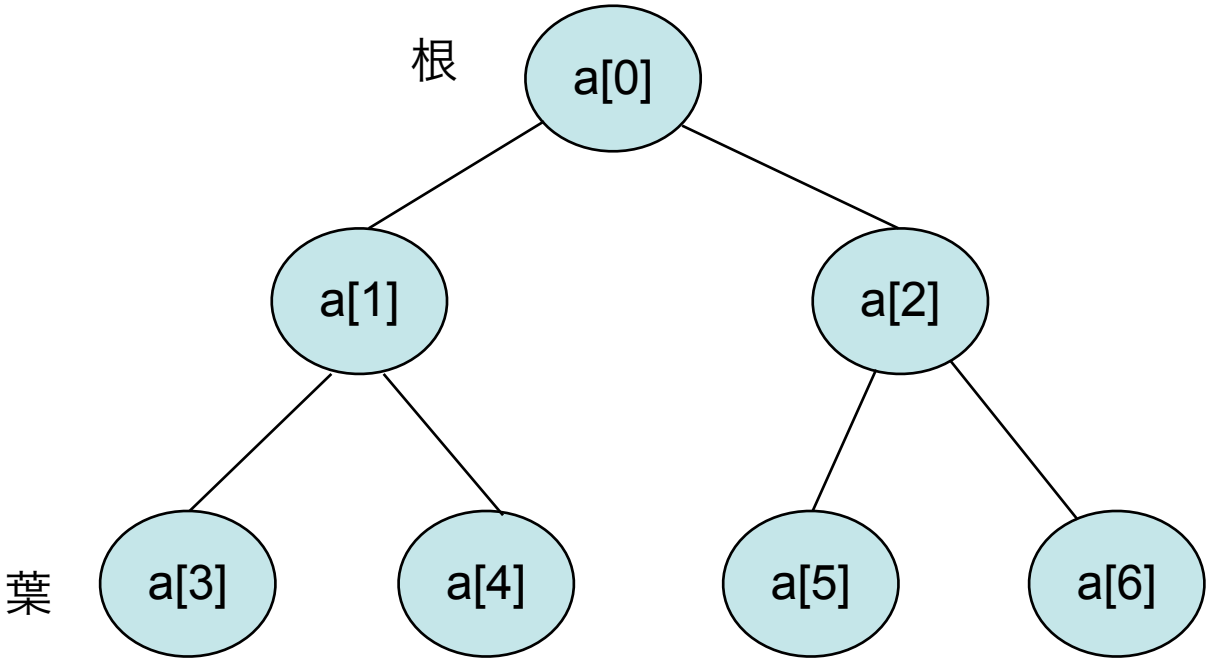
完全2分木



ヒープ

59

完全2分木の配列化



配列の関係

a[i]の親：a[(i-1)/2]

a[i]の左の子:a[i*2+1]

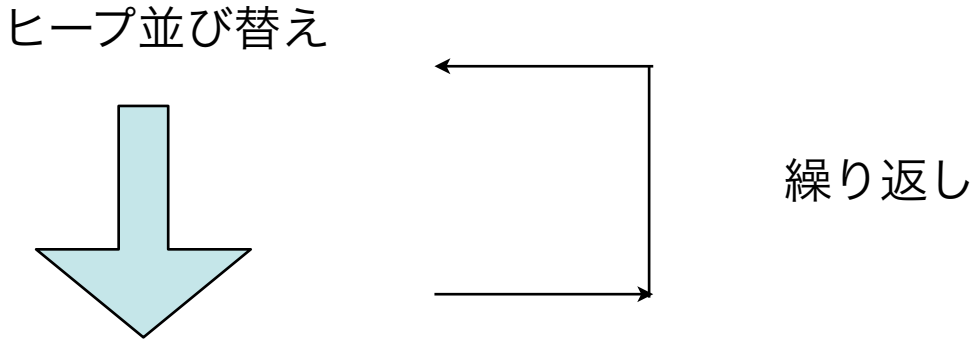
a[i]の右の子:a[i*2+2]

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
9	8	7	6	5	4	3

60

ヒープソートの原理

特徴：ヒープの根には最大値がある



ヒープソート

ダウンヒープ手順

(1)根を取り出す

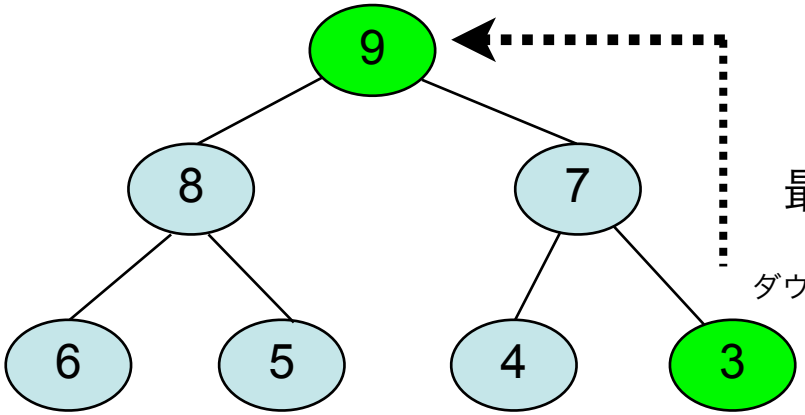
(2)最後の要素を根に移動する

(3)根に着目しその値が、大きい方の子より小さければ交換

子の方が小さくなるか葉に到達するまで繰り返す

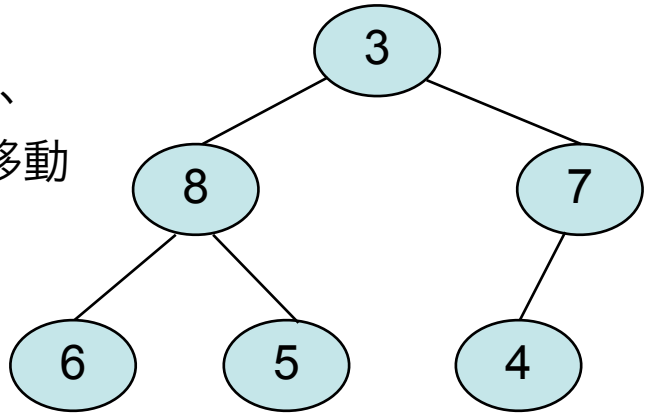
61

ヒープソートの手順１



根を取りだし、
最後の要素を移動

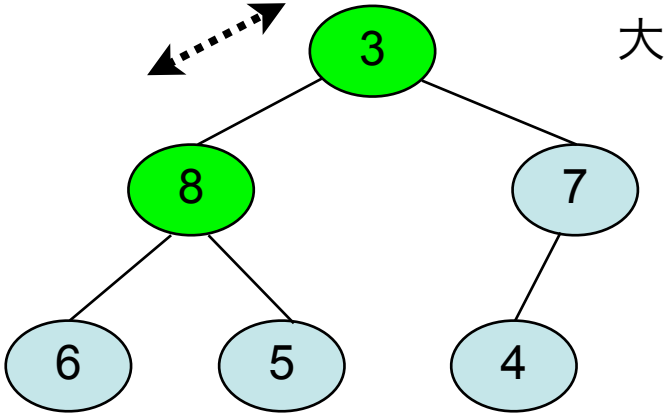
ダウンヒープ１
開始

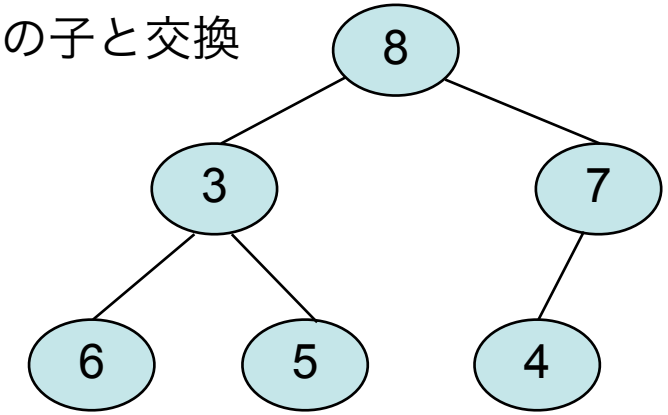


a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
9	8	7	6	5	4	3

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
3	8	7	6	5	4	9

大きい値を持つ左の子と交換



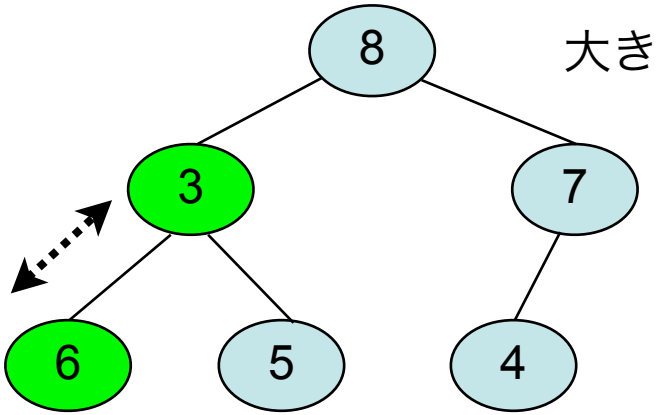


a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
3	8	7	6	5	4	9

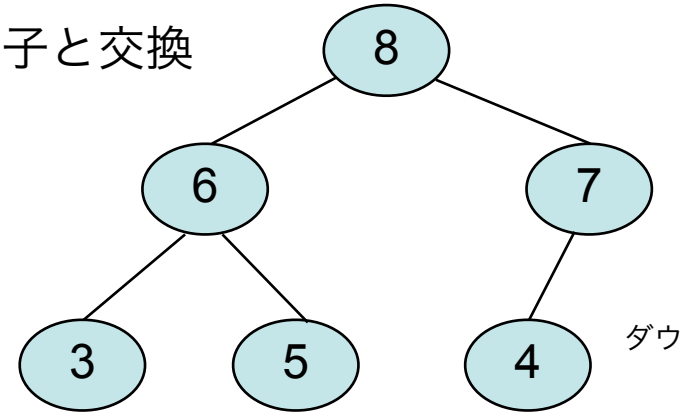
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
8	3	7	6	5	4	9

62

ヒープソートの手順２



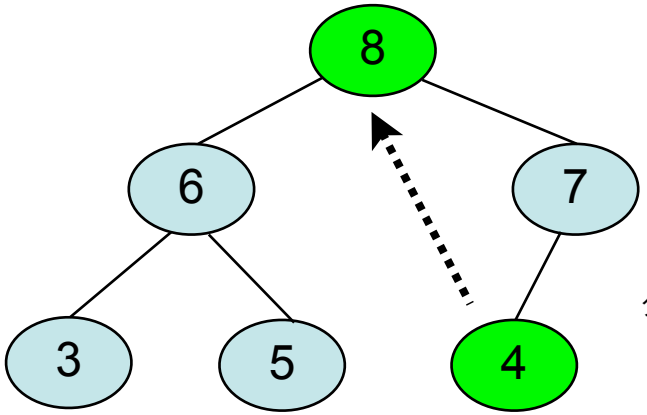
大きい値を持つ左の子と交換



ダウンヒープ１
終了

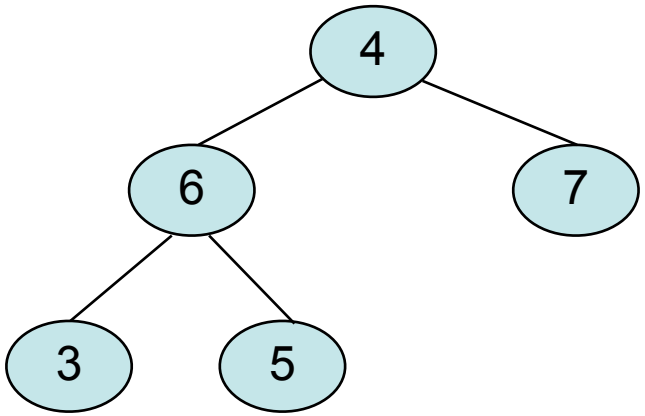
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
8	3	7	6	5	4	9

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
8	6	7	3	5	4	9



根を取りだし、
最後の要素を移動

ダウンヒープ２
開始

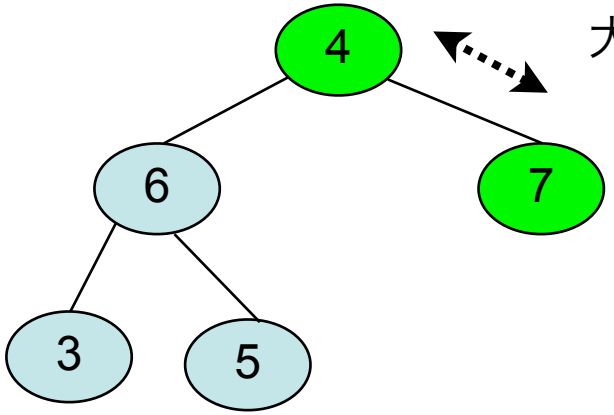


a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
8	6	7	3	5	4	9

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
4	6	7	3	5	8	9

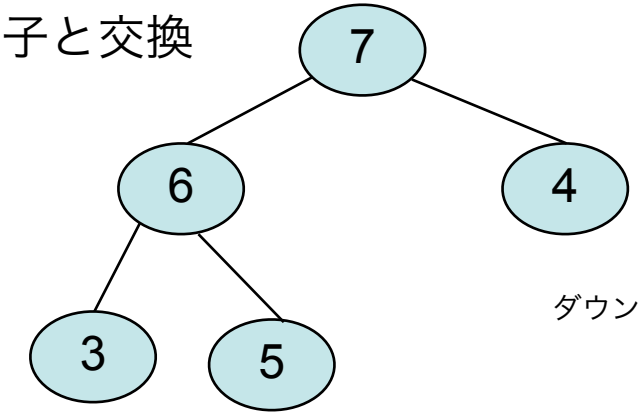
63

ヒープソートの手順３



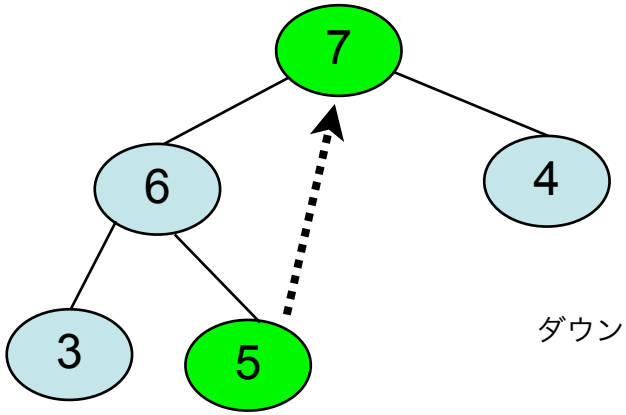
大きい値を持つ右の子と交換

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
4	6	7	3	5	8	9



ダウンヒープ２
終了

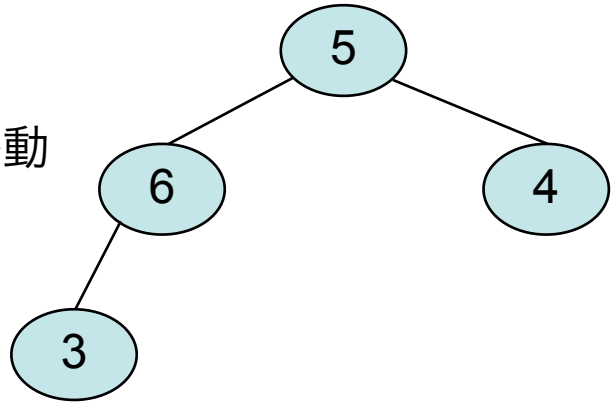
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
7	6	4	3	5	8	9



ダウンヒープ３
開始

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
7	6	4	3	5	8	9

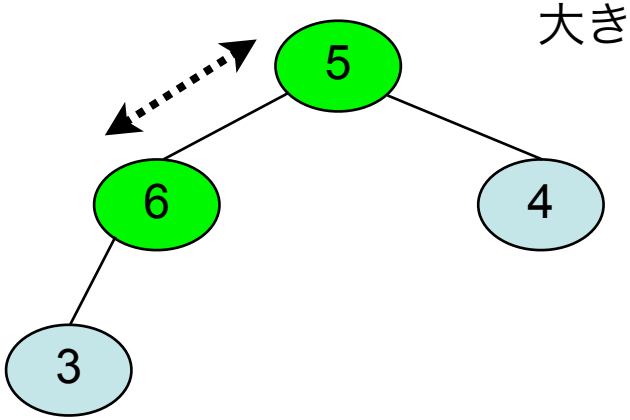
根を取りだし、
最後の要素を移動



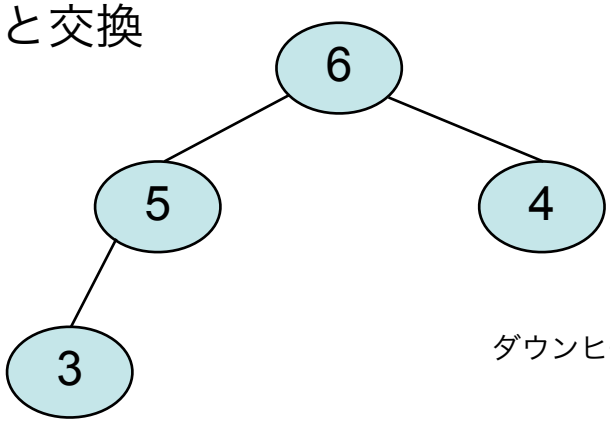
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
5	6	4	3	7	8	9

ヒープソートの手順４

大きい値を持つ左の子と交換



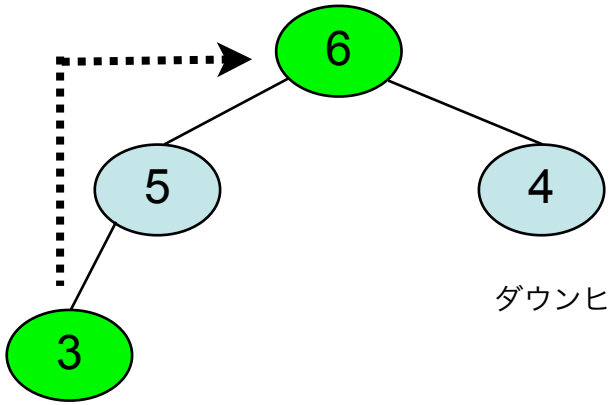
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
5	6	4	3	7	8	9



ダウンヒープ３
終了

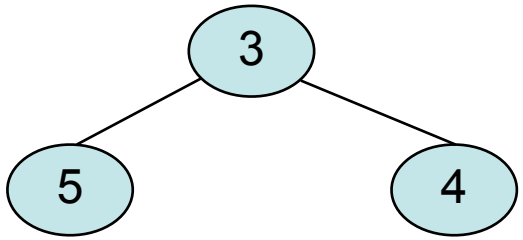
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
6	5	4	3	7	8	9

根を取りだし、
最後の要素を移動



ダウンヒープ４
開始

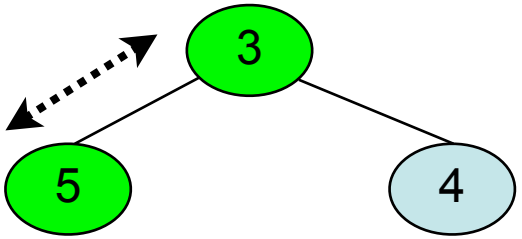
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
6	5	4	3	7	8	9



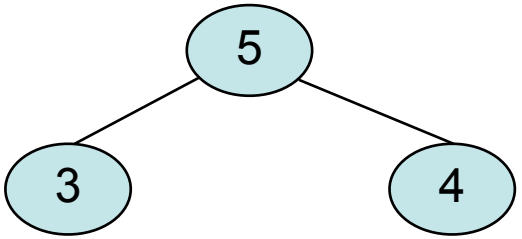
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
3	5	4	6	7	8	9

65

ヒープソートの手順５



子と交換

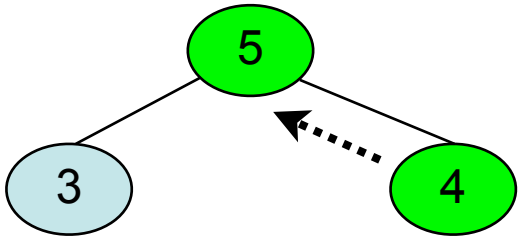


ダウンヒープ４
終了

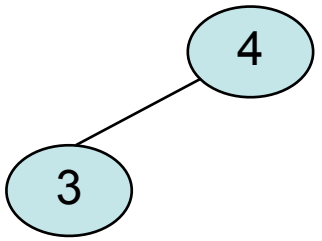
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
3	5	4	6	7	8	9

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
5	4	4	6	7	8	9

根を取りだし、
最後の要素を移動



ダウンヒープ５
開始



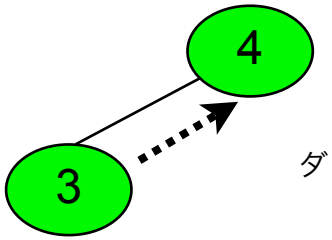
ダウンヒープ５
終了

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
5	3	4	6	7	8	9

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
4	3	5	6	7	8	9

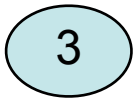
66

ヒープソートの手順６



ダウンヒープ６
開始

根を取りだし、
最後の要素を移動



ダウンヒープ６
終了

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
4	3	5	6	7	8	9

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
3	4	5	6	7	8	9

終了

ダウンヒープ：a[i]とa[0]を交換し、a[0]からa[i-1]までを対象にヒープする

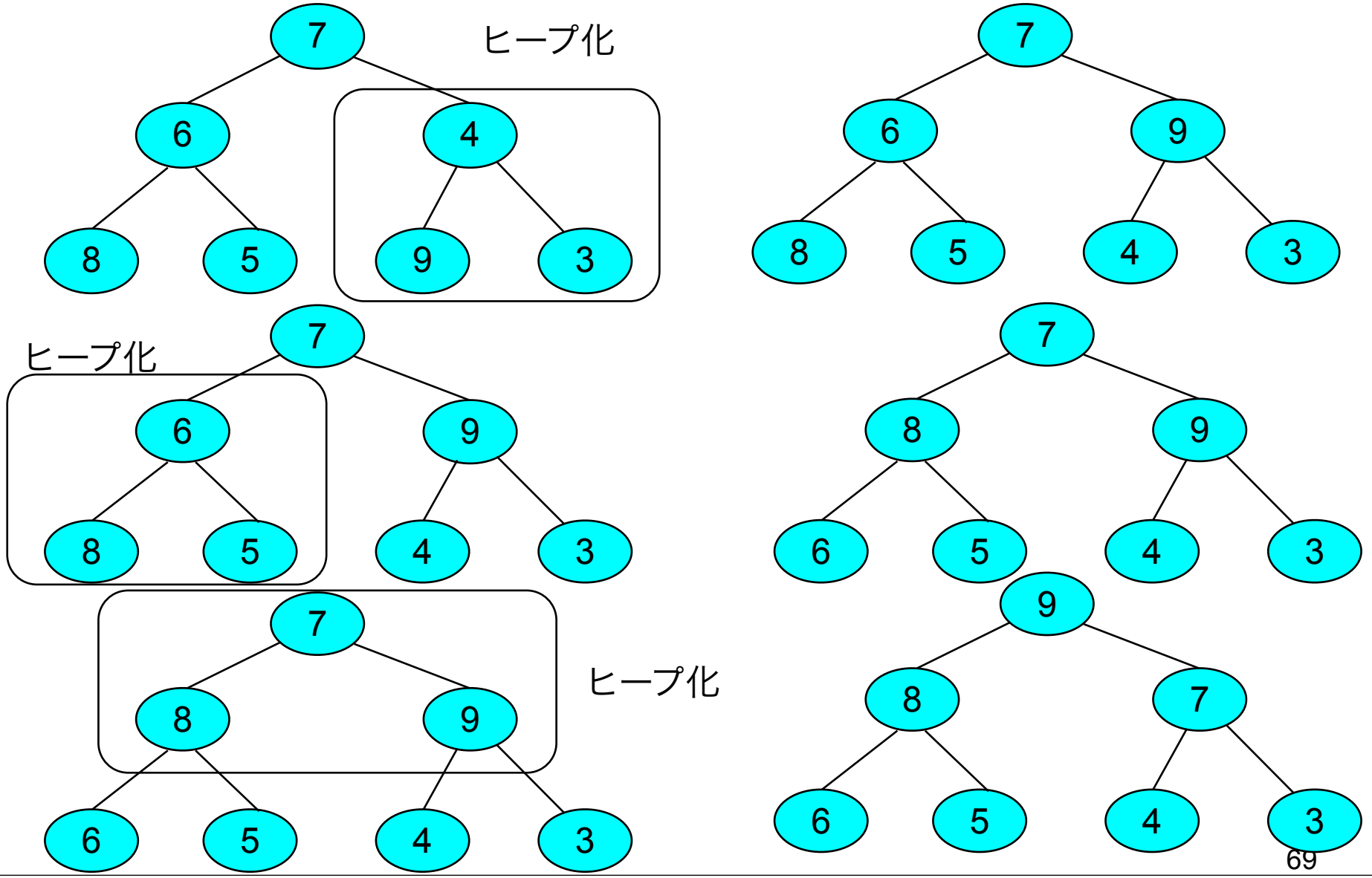
67

ヒープソートを適用する条件

- ヒープソートを適用するためには
- 適用する配列がヒープ化されている
  - 配列の初期化
    - ダウンヒープを繰り返すことによって初期化する
      - 葉を含む部分木から順にヒープ化
      - 根を含むまで繰り返す
        - » ヒープ化終了

68

配列のヒープ化(初期化)



69

ヒープソートプログラム1(メイン)

```
#include <stdio.h>
#define swap(type,x,y) do {type t=x; x=y; y=t;} while(0)
#define NUM 5
void downheap(int a[],int left, int right); /* 関数プロトタイプ  */
void heapsort(int a[], int n);
int count0=0,count1=0;count2=0;/* count0:比較,count1:交換,count2:挿入  */
int main(void)
{
        int     i;
        int             x[NUM];

        printf(" Input integer number %d times ¥n",NUM);
        for (i=0;i<NUM;i++)       {
                printf("x[%d]:",i);
                scanf("%d",&x[i]);
        }
        heapsort(x,NUM);
        printf("Sorting is finished ¥n");
        for (i=0;i<NUM;i++)
                printf("x[%d] =%d¥n",i,x[i]);

        printf("Number of comparison=%d¥n",count0);
        printf("Number of swap=%d¥n",count1);
        printf("Number of insertion=%d¥n",count2);
        return(0);
}
```

ヒープソートプログラム2(関数)

```
void downheap(int a[],int left, int right) /* ダウンヒープ関数  */
                                    /* leftからrightまでをヒープ化  */
                                    /* 前提: a[left+1]～a[right]はヒープ済み */
{
        int     temp=a[left];
        int     child;
        int     parent;

        for (parent=left;parent<(right+1)/2;parent=child) {
                int  cl=parent*2+1;        /* left child 左の子  */
                int  cr=cl+1;              /* right child 右の子  */
                if (cr<=right && a[cr]>a[cl]) { child=cr; count0+=2; }
                else { child=cl; count0+=2; }  /* 子の大きい方を選択  */
                if (temp>=a[child]) break;     /* 子が小さい場合  ループから抜ける  */
                a[parent]=a[child];            /* 子の値を親に代入  */
        }
        a[parent]=temp;     /* a[left]をヒープが成立する位置に挿入  */
        count2++;
}
```


71

ヒープソートプログラム3(関数)

```
void heapsort(int a[], int n)
{
        int     i;

        for (i=(n-1)/2;i>=0;i--)
                downheap(a,i,n-1);  /* 配列初期化  */

        for (i=n-1;i>0;i--)       {  /* ダウンヒープの繰り返し  */
                swap(int, a[0],a[i]); count1++;
                downheap(a,0,i-1);
        }
}
```


72

ソートプログラム実行結果

```
Input integer number 5 times
x[0]:60
x[1]:75
x[2]:70
x[3]:56
x[4]:52
Sorting is finished
x[0] =52
x[1] =56
x[2] =60
x[3] =70
x[4] =75
Number of comparison=12
Number of swap=4
Number of insertion=7
```


73

ヒープソートの計算量

データn個のソート比較回数

- n個のデータのヒープ段数：k
  - $n=2^k$
  - $k=\log_2 n$
  - 1回のヒープ化でk-1回の比較
- k-1回の比較をn個のデータ分行う

$n*(k-1)=n*(\log_2 n-1)$ ⇒ $O(n\log_2 n)$

74

ヒープソートの特徴

- まとめ
  - 各ステップで最大値を求める操作を繰り返す
  - 途中で得られる大小関係をヒープというデータ構造に蓄積する
  - ヒープ後には最大値を求める計算量は1
  - ヒープソートのオーダは $n \log_2 n$

75

処理時間計測1（メイン）

1. 2000個乱数を発生させ、それをソートするプログラムを2000回実行する。
2. バブルソート、単純選択法、挿入法、クイックソート、ヒープソートで処理時間を比較する。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define NUM 2000    /* ソートするデータの個数　変更可能　*/
#define NUM1 2000   /* ソートする繰り返し回数　変更可能　*/
#define swap(type,x,y) do {type t=x; x=y; y=t;} while(0)

void bubble(int a[], int n);
void selsort(int a[], int n);
void insertion(int a[], int n);
void quick(int a[],int left, int right);
void downheap(int a[],int left, int right);
void heapsort(int a[], int n);
```


76

参考：処理時間計測2（メイン）

```
int main(void)
{
        int       i,j;
        int       x[NUM];
        double    temp;
        double    dt0=0.0,dt1=0.0,dt2=0.0,dt3=0.0,dt4=0.0; /* ソート積算時間  */
        time_t    start,end;

        srand(time(NULL));

        for (j=1;j<=NUM1;j++)       {
                for (i=0;i<NUM;i++)          {  /* 乱数発生  */
                        temp=(double)rand()/(double)RAND_MAX;
                        x[i]=(int)(temp*1000.0);
                        }
                start=clock();    /* ソート時間の計測  */
                bubble(x,NUM);
                end=clock();
                dt0=dt0+(double)(end - start) / CLOCKS_PER_SEC;
        }

               :
               :
               (省略)
               :
               :
                        /* ソート時間　結果表示 */
        printf("Running time bsort=%lf (sec), ssort=%lf (sec),
             isort=%lf (sec)¥n",dt0,dt1,dt2);
        printf("Running time qsort=%lf (sec), hsort=%lf (sec) ¥n",dt3,dt4);

        return(0);
}
```


77

参考：処理時間計測3(関数)

```
void bubble(int a[], int n)
{
      int           i,j;

      for (i=0;i<n-1;i++) {
          for (j=n-1;j>i;j--) {
              if (a[j-1]>a[j]) {
                    swap (int, a[j-1],a[j]);
              }
           }
      }
}
```

```
void selsort(int a[], int n)
{
    int   i,j,min;

          for (i=0;i<=n-1;i++)  {
              min=i;
              for (j=i+1;j<=n-1;j++)    {
                    if (a[j]<a[min]) {
                        min=j;
                    }
              swap (int, a[i],a[min]);
              }
          }
}
```


78

参考：処理時間計測3(関数)

```
void insertion(int a[], int n)
{
          int       i,j,tmp;
          for (i=1;i<=n-1;i++){
              tmp=a[i];
              j=i;
              while ((a[j-1]>tmp) && (j>0)) {
                    a[j]=a[j-1];
                    j--;
                    }
          a[j]=tmp;
          }
}
```

```
void quick(int a[],int left, int right)
{
      int    pl=left;
      int    pr=right;

      int    x=a[(pl+pr)/2]; /* pivot */

      do {
          while (a[pl]<x) { pl++; }
          while (a[pr]>x) { pr--; }
          if (pl<=pr) {
                    swap(int, a[pl],a[pr]);
                    pl++;
                    pr--;
                    }
      } while (pl<=pr);

      if (left<pr) quick(a,left,pr);
      if (pl<right) quick(a,pl,right);
}
```


79

参考：処理時間計測5(関数)

```
void downheap(int a[],int left, int right)
{
      int    temp=a[left];
      int    child;
      int    parent;

      for (parent=left;parent<(right+1)/
           2;parent=child) {
          int cl=parent*2+1;  /* left child */
          int cr=cl+1;        /* right child */

      if (cr<=right && a[cr]>a[cl]) child=cr;
      else child=cl;
      if (temp>=a[child])
          break;
          a[parent]=a[child];
      }
      a[parent]=temp;
}
```

```
void heapsort(int a[], int n)
{
          int       i;

          for (i=(n-1)/2;i>=0;i--)
                    downheap(a,i,n-1);

          for (i=n-1;i>0;i--) {
                    swap(int, a[0],a[i]);
                    downheap(a,0,i-1);
          }
}
```


80

参考：処理時間計測（実行結果）

```
Running time bsort=40.550000 (sec), ssort=50.433333 (sec),
     isort=13.183333 (sec)
Running time qsort=0.500000 (sec), hsort=0.833333 (sec)
```

この例では

クイックソート＜ヒープソート＜単純挿入法＜

単純選択法＜単純交換法

となった。

81

課題問題（レポート提出要）

課題の目的：各種ソートアルゴリズムの計算時間の違いを理解する

- 処理時間計測プログラムのソースコードを入力し、実行する。
- 入力データ数を変化させて計算時間を確認する（入力データ数の変更等のプログラム修正は各自の設計に基づいて行うこと）
- 各種ソートの計算量とオーダを考察する

レポートを作成し7/1に提出すること

レポートのフォーマットは自由とする

82

まとめ（ソート）

	最良の場合	最悪の場合	平均的の場合
バブルソート	$O(n^2)$	$O(n^2)$	$O(n^2)$
単純選択ソート	$O(n^2)$	$O(n^2)$	$O(n^2)$
挿入法	$O(n)$	$O(n^2)$	$O(n^2)$
クイックソート	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$
ヒープソート	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$

83