Dhieka Avrilia Lantana

ANALISIS ALGORITMA *BINARY SEARCH*

Metode *Binary search*

Binary search merupakan salah satu algoritma untuk melalukan pencarian pada array yang sudah terurut. Jika kita tidak mengetahui informasi bagaimana integer dalam array, maka penggunaan *binary search* akan menjadi tidak efisien, kita harus melakukan sorting terlebih dahulu atau menggunakan metode lain yaitu *linear search*. Namun jika kita telah mengetahui integer dalam array terorganisasi baik secara menaik atau menurun, maka bisa dengan cepat menggunakan algoritma *binary search*. Adapun ide dasar *binary search* yaitu memulai pencarian dengan membagi dua ruang pencarian. Misalnya kita memiliki array A, dan kita ingin menemukan lokasi dari spesifik target integer K dalam array. Ada 3 kemungkinan kondisi pada *binary search* yaitu:

1. Jika data target K langsung di temukan, maka proses pembagian ruangan berhenti. Kemudian print out indeks data elemen pada array.
2. Jika data target K < A[middle], maka pencarian dapat dibatasi hanya dengan melakukan pencarian pada sisi kiri array dari A[middle]. Seluruh elemen yang berada di sebelah kanan dapat di abaikan.
3. Jika data target K > A[middle], maka akan lebih cepat jika pencarian di batasi hanya pada bagian sebelah kanan saja.
4. Jika seluruh data telah di cari namun tidak ada, maka diberi nilai seperti -1.

Dibawah ini merupakan salah satu version program *binary search*.

```
Kamus
   Conts N :
   Type t=array[0 … N] of integer
   val, left, right, mid : Integer

Algoritma
int binarySearch(LIST t[], int n, int val)
{
   int left, right, mid;

   left = 0; right = n-1;
   while(left<=right) {
      mid=(left+right)/2;
      if (val<t[mid].key)
         right=mid-1;
      else if (val>t[mid].key)
         left=mid+1;
      else
         return mid; /* found */
   }
   return -1; /* not found */
}
```

Analisis Kompleksitas

Contoh data yang sudah terurut :

A	1	3	8	13	17	18	25	57
indeks	0	1	2	3	4	5	6	7

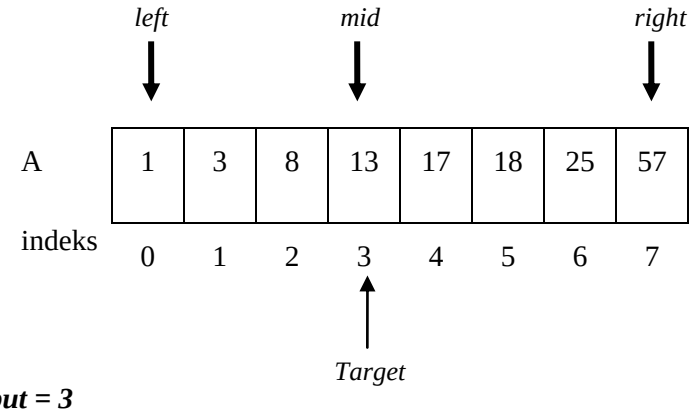
Kasus 1 : cari = 13

Left = 0

Right = 7

Loop (1) : mid = (left +right) div 2 = (0 + 7) div 2 = 3

t [mid] = A [3] = 13, pada loop pertama data langsung ditemukan



Output = 3

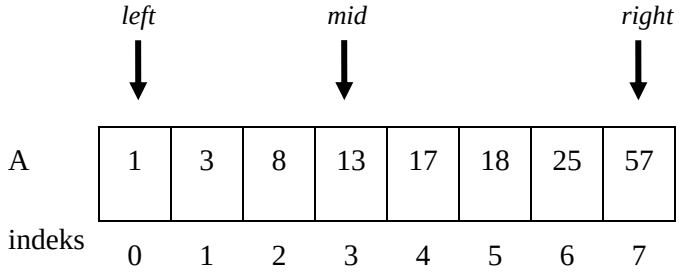
Kasus 2 : cari = 17

Left = 0

Right = 7

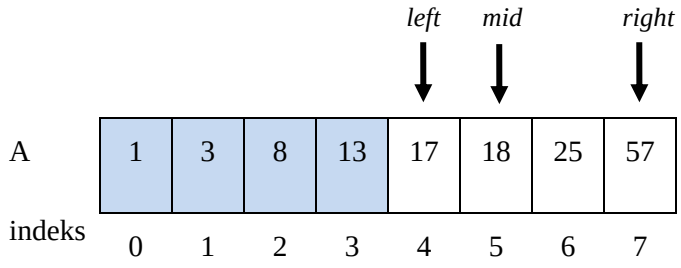
Loop (1) : mid = (left +right) div 2 = (0 + 7) div 2 = 3

t [mid] = A [3] = 13 < val = 17, data belum ditemukan, berarti left = mid + 1 = 3 + 1 = 4



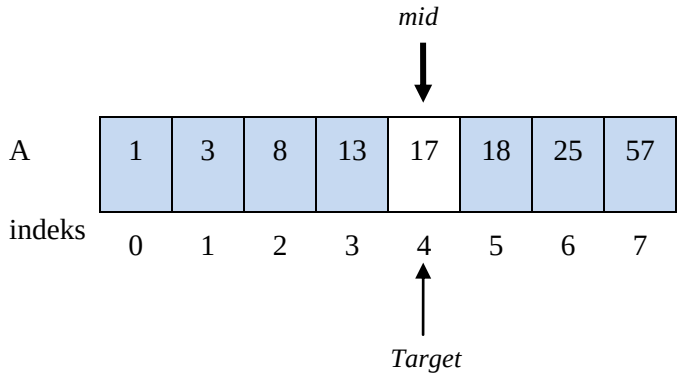
Loop (2) : mid = (left +right) div 2 = (4 + 7) div 2 = 5

t [mid] = A [5] = 18 > val = 17, data belum ditemukan, berarti right = mid - 1 = 5 - 1 = 4



Loop (3) : mid = (left +right) div 2 = (4 + 4) div 2 = 4

t [mid] = A [4] = 17 = val = 17, data ditemukan setelah loop ketiga



Output = 4

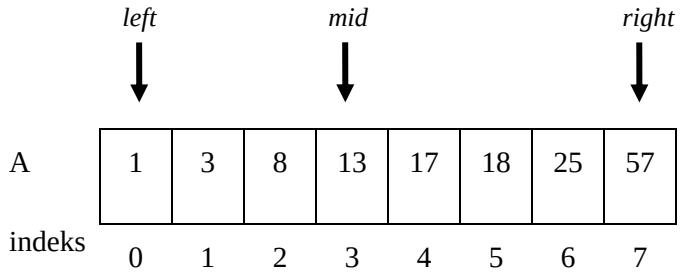
Kasus 3 : cari = 2

Left = 0

Right = 7

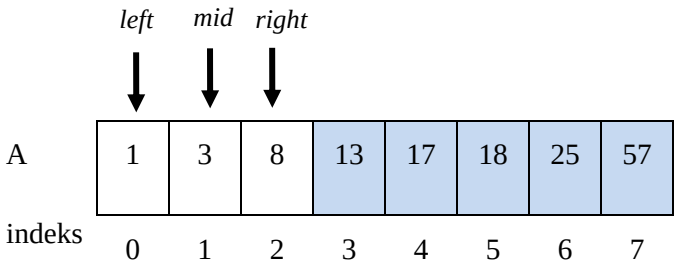
Loop (1) : mid = (left + right) div 2 = (0 + 7) div 2 = 3

t [mid] = A [3] = 13 > val = 2, data belum ditemukan, berarti right = mid - 1 = 3 - 1 = 2



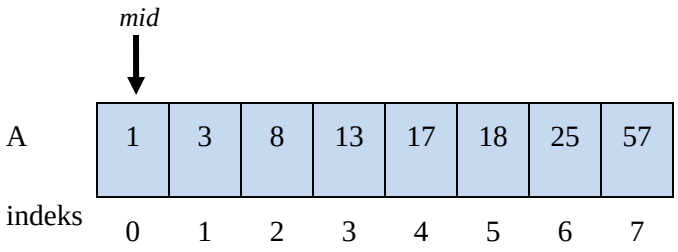
Loop (2) : mid = (left +right) div 2 = (0 + 2) div 2 =1

t [mid] = A [1] = 3 > val = 2, data belum ditemukan, berarti right = mid - 1 = 1 - 1 = 0



Loop (3) : mid = (left +right) div 2 = (0 + 0) div 2 = 0

t [mid] = A [0] = 1 < val = 2, data belum ditemukan, left = mid+1 = 0 + 1 = 1



Loop (4) : mid = (left +right) div 2 = (1 + 0) div 2 = 0

Tidak bisa dijalankan karena 1 lebih dari 0 sehingga loop berhenti di loop ketiga. Sehingga data tidak dapat ditemukan

Output = -1

Sekarang mari kita analisis metode binary search untuk menentukan kompleksitasnya. Ketika jumlah elemen dalam array 8:

Ketika n=8, Binary Search dijalankan dengan mereduksi ukuran menjadi 4

Ketika n=4, Binary Search dijalankan dengan mereduksi ukuran menjadi 2

Ketika n=2, Binary Search dijalankan dengan mereduksi ukuran menjadi 1

Dapat kita lihat bahwa binary search dipanggil sebanyak tiga kali (3 elemen dalam array yang dieksekusi) untuk n = 8.

Sehingga didapat $8 = 2^3$ atau secara general kita katakan $n = 2^k$. ketika kita mengeksekusi *x* pencarian, "while loop" juga dieksekusi sebanyak *x* kali dan n di reduksi ukurannya menjadi

1. Pada contoh penerapan di atas, dapat disimpulkan jumlah maksimum total operasi yang dilakukan adalah sebanyak 3. Nilai dari k dapat dinotasikan menjadi

$$\mathbf{2^k = n \text{ sehingga } } \boldsymbol{k} \mathbf{= \log_2 n}$$

Jumlah operasi yang dilakukan untuk mencari k adalah sebanyak = 3 log n. pada kasus terburuk, yaitu kasus dimana *k* tidak terdapat dalam array adalah

$$T(n) = \mathbf{\log_2 n}$$

Misalnya jika kita memiliki array sebanyak 1024 elemen, kasus terburuk menghasilkan pembandingan array sebanyak $\log_2 1024 = 10$ kali. Bandingkan dengan *linear search* yang pada kasus terburuknya melakukan pembandingan sebanyak 1024. Tentunya algoritma *binary search* menjadi lebih cepat. Namun jika data yang ada merupakan data yang tidak terurut maka akan jauh lebih cepat jika menggunakan *linear search*. Jika menggunakan *binary search* maka akan ada cost tambahan yaitu mengurutkan array terlebih dahulu. Algoritma *binary search* biasa di gunakan untuk *database*. Pada *database* tidak perlu ada algoritma sorting karena pada *database* sendiri sudah disediakan fungsi sorting baik untuk menaik atau menurun.

Binary Search Tree

Binary Tree ini memiliki sifat dimana semua left child harus lebih kecil dari pada right child dan parentnya. Semua right child juga harus lebih besar dari left child serta parentnya. Binary search tree dibuat untuk mengatasi kelemahan pada binary tree biasa, yaitu kesulitan dalam searching / pencarian node tertentu dalam binary tree.

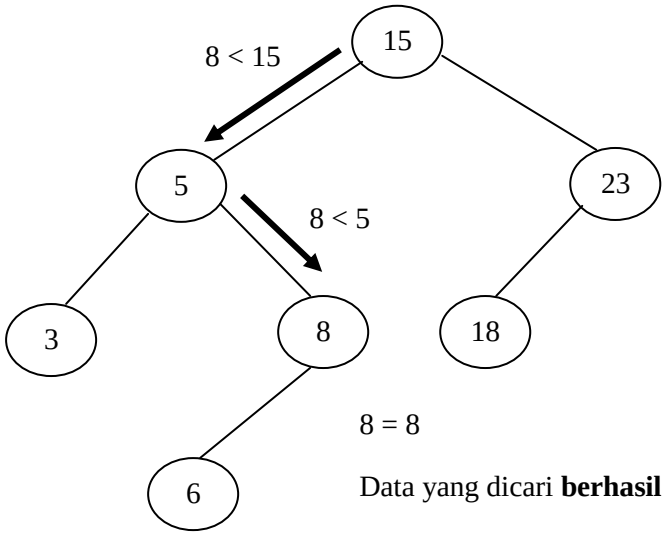
Algoritma

```
Algorithm searchBST (root, targetKey)
  If (empty tree)
      return null   *not found
  End if
  If (targetKey < root)
      return searchBST (left subtree, targetKey)
   else if (targetKey > root)
      return searchBST (right subtree, targetKey)
  else
      return root *found target
  end if
end searchBST
```

Contoh Binary Search tree

Proses pencarian sukses

Search(7)

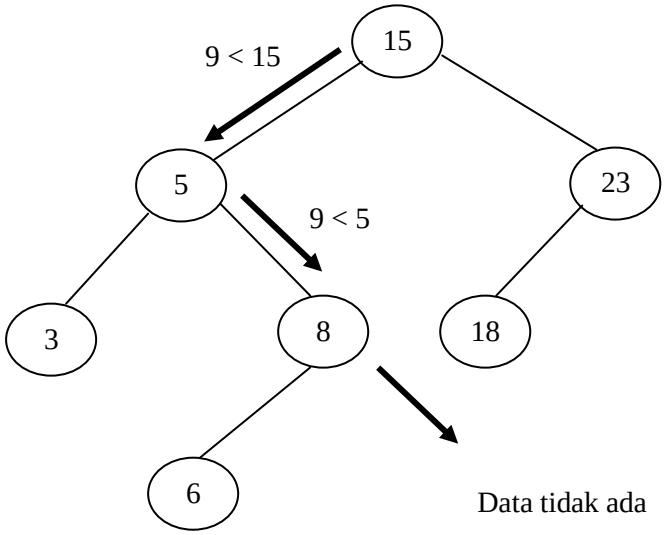


8 = 8

Data yang dicari **berhasil** ditemukan

Proses pencarian gagal

Search(9)



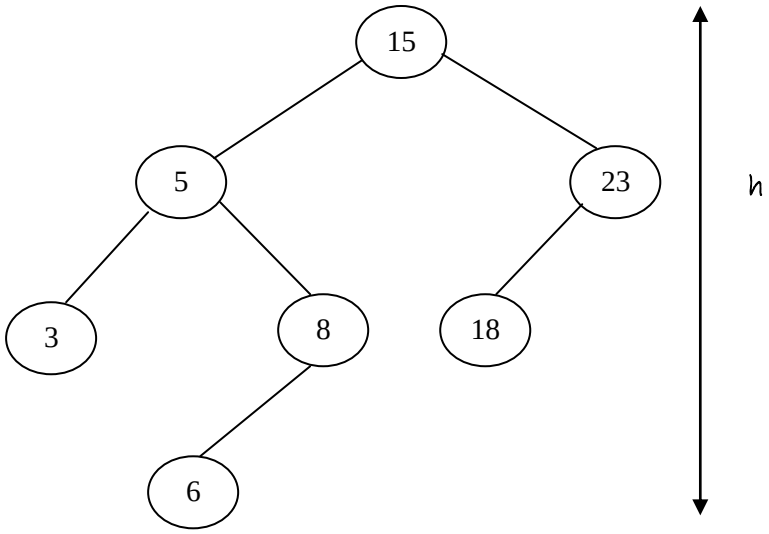
Data tidak ada

Data yang dicari **GAGAL** ditemukan

Analisis Kompleksitas Binary Search Tree

Kelemahan yang sangat mendasar pada Binary search tree adalah elemen-elemen pada tree yang harus berurut. Binary Tree yang tidak *balance* dapat membuat seluruh operasi memiliki kompleksitas running time O(n) pada kondisi worst case. Sedangkan pada kondisi

Binary Tree yang balanced maka waktu rata-ratanya adalah log n. Proses pencarian pada tree sangat bergantung pada ketinggian suatu tree. Pada contoh kasus diatas ketinggian tree adalah 3.



Maksimum operasi perbandingan jika data gagal ditemukan adalah sebanyak 3 kali. Jika data yang dicari berada di root (Target=15), best casenya adalah 1 kali pembandingan.

Daftar Pustaka

Aslam & Fell.*Analysis of Algorithms: Running Time.* 2005. CSU 2000 discrete Structures Fall 2005.