Penerapan BFS dan DFS dalam *Garbage Collection*

Nugroho Satriyanto – 13514038
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung
Bandung, Indonesia
13514038@std.stei.itb.ac.id

***Abstract*—*Garbage collection* adalah proses yang sangat penting dalam pengembangan suatu bahasa pemrograman. Hal itu dikarenakan *garbage collection* dapat meningkatkan performansi program dalam hal ruang dan waktu. Terdapat beberapa algoritma untuk implementasi *garbage collection*, salah satunya adalah Copying *Garbage Collection* yang dapat diimplementasikan dengan BFS dan DFS**

Keywords—BFS; DFS; garbage; heap; stack; memori

I. PENDAHULUAN

Garbage collection adalah fitur yang ada pada compiler untuk mengumpulkan dan memisahkan antara memori yang masih terpakai, dan membuang memori yang tidak terpakai lagi untuk menambah ruang memori yang tersedia. *Garbage collection* haruslah efisien mengingat hal tersebut sangat penting dan dipanggil oleh sistem operasi secara teratur. Memori sampah yang sudah dipilah oleh *garbage collection* compiler nantinya akan diolah oleh sistem operasi untuk dibuang.

Beberapa bahasa pemrograman menyediakan *garbage collection* otomatis dimana pembuat program tidak perlu menyatakan dealokasi secara manual. Beberapa bahasa yang tidak perlu menyatakan dealokasi adalah java, C#. Bahasa yang perlu menyatakan dealokasi tetapi memiliki implementasi *garbage collection* sendiri yang berjalan untuk kasus tertentu saja, misalnya C dan C++ yang mempunyai *garbage collection* tetapi harus melakukan dealokasi manual untuk objek pointer. Beberapa bahasa pemrograman juga dapat memberikan akses ke programmer untuk menonaktifkan *garbage collection*, seperti bahasa D.

II. LANDASAN TEORI

2.1 Algoritma BFS dan DFS

Graf adalah himpunan dari objek-objek yang dinamakan titik, simpul, atau sudut dihubungkan oleh penghubung yang dinamakan garis atau sisi. Dalam pengolahan informasi pada graf, diperlukan suatu cara untuk menelusuri semua simpul graf dan mengolah informasi yang terdapat pada graf atau biasa disebut transversal. Transversal dasar pada graf ada tiga, yaitu,

a. Preorder: melakukan pemrosesan terhadap simpul akar lalu tetangga-tetangganya.
b. Postorder: melakukan pemrosesan terhadap tetangga-tetangganya dahulu lalu simpul akar.
c. Inorder (untuk pohon biner): melakukan pemrosesan upapohon kanan, akar, lalu upapohon kiri secara berurutan.

Selain ketiga transversal tersebut, terdapat pula transversal lain yaitu *Breadth First Search* dan *Depth First Search*.

2.1.1 *Breadth First Search* (BFS)

Breadth-first search atau pencarian melebar adalah pencarian dengan urutan sebagai berikut apabila pencarian dimulai dari simpul v:

1. Mengunjungi simpul v.
2. Mengunjungi semua simpul yang bertetangga dengan simpul v.
3. Mengunjungi simpul yang belum dikunjungi dan bertetangga dengan simpul-simpul yang tadi dikunjungi, demikian seterusnya..

Algoritma ini mirip dengan algoritma Dijkstra untuk mencari jalur terdekat dalam labirin yang juga direpresentasikan dalam graf, tetapi dengan semua simpulnya memiliki bobot yang sama.

Pencarian BFS dapat direpresentasikan dalam antrian (*queue*) dengan elemen *queue* menunjuk pada simpul yang dikunjungi. Mulanya elemen antrian hanya berisi v, lalu jika v telah selesai dikunjungi, maka selenjutnya mengunjungi semua simpul yang bertetangga dengan v. Berikut adalah algoritma sederhananya, dengan menganggap semua simpul memiliki penanda yang menunjukkan apakah simpul tersebut sudah dikunjungi.

```
hapus semua tanda
masukkan v ke dalam queue
while (queue tidak kosong)
    tandai elemen pertama queue
    hapus elemen pertama queue
    masukkan semua simpul yang bertetangga dengan elemen pertama queue ke dalam queue
```

Yang perlu diperhatikan adalah menghapus simpul yang telah selesai dikunjungi dari antrian dan menambahkan simpul yang bertetangga dengannya sebagai elemen antrian terakhir.

2.1.2 Depth First Search

Depth-first search atau pencarian mendalam adalah pencarian dengan urutan sebagai berikut apabila pencarian dimulai dari simpul v:

1. Mengunjungi simpul v.
2. Mengunjungi simpul w yang bertetangga dengan simpul v.
3. Mengulangi DFS untuk simpul w.
4. Ketika mencapai simpul u sedemikian sehingga semua simpul yang bertetangga dengannya telah dikunjungi, pencarian dirunut-balik (backtrack) ke simpul terakhir yang dikunjungi sebelumnya dan mempunyai simpul w yang belum dikunjungi.
5. Pencarian berakhir bila tidak ada lagi simpul yang belum dikunjungi yang dapat dicapai dari simpul yang telah dikunjungi.

Pencarian DFS ini mirip dengan pencarian postorder. Tetapi untuk menghindari perulangan terus-menerus yang mungkin diakibatkan karena graf bersifat siklik, diperlukan semacam penanda untuk setiap simpul yang telah dikunjungi, sehingga DFS tidak diulangi untuk simpul yang telah dikunjungi.

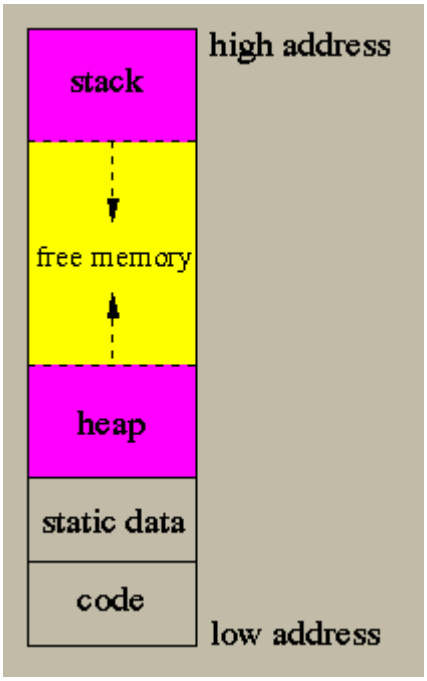
Jika BFS bisa direpresentasikan dengan antrian, maka DFS bisa direpresentasikan dengan tumpukan (stack), dengan cara

hapus semua tanda
masukkan v dalam stack
while (stack tidak kosong)
    hapus stack pertama dan tandai sudah dikunjungi
    bangkitkan semua tetangganya dan masukkan dalam stack

2.2 Manajemen Memori File Eksekutabel

Setelah melakukan kompilasi pada suatu *source code* akan tercipta suatu file eksekutabel. File eksekutabel tersebut menyimpan semua informasi yang diperlukan oleh program saat dijalankan.

Data-data yang perlu disimpan di antaranya adalah kode program, data statis, data dinamis, dan *stack* untuk menyimpan alamat-alamat program. Cara dalam merepresentasikannya dapat berbeda-beda antar satu *compiler* dengan *compiler* lain. Tetapi struktur dasarnya adalah seperti pada gambar di bawah ini.



gambar 1: struktur file eksekutabel (https://lambda.uta.edu/cse5317/notes/node45.html)

Gambar di atas menunjukkan struktur dari file eksekutabel. Data statis adalah data yang tidak mungkin berubah ukurannya, misalnya suatu variabel yang langsung didefinisikan, misal int i=2. Variabel i akan langsung disimpan nilainya pada data statis. Sementara variabel yang belum didefinisikan nilainya, misal int a, hanya akan dihitung jumlah memori yang ditempatinya dan dimasukkan ke dalam header suatu file. Variabel yang tidak langsung diinisialisasi ini nantinya akan diinisialisasi sebagai nol atau lainnya sesuai karakteristik *compiler*.

Metode penyimpanan yang tidak pasti seperti pointer pada C dan C++ atau array yang belum terinisialisasi pada Java, akan disimpan pada *heap* dan baru akan disimpan setelah program dijalankan.

Jika program menggunakan banyak memori dinamis dan tidak didealokasikan, terutama untuk C dan C++ yang tidak memiliki *garbage collection* untuk pointer, ruang *heap* akan habis dan program akan meminta tambahan *heap*. Keadaan ini sangatlah tidak menguntungkan karena bagian stack harus dipindahkan ke lokasi yang lebih tinggi, yang berarti menyalin semua data stack lalu memindahkannya satu per satu. Untuk itu diperlukan cara untuk mendealokasikan memori dinamis seperti free pada C dan delete pada C++.

Untuk menentukan alamat memori mana yang termasuk *garbage*, program menyimpan suatu pointer yang saling berhubungan antar alamat memori yang hidup (masih digunakan). Alamat memori yang tidak dapat diraih oleh rantai pointer tersebut akan dianggap sebagai *garbage* dan dibuang jika *heap* penuh. Walau begitu, *garbage collection* tidak akan bisa menentukan memori yang dialokasi dan terdapat dalam rantai pointer, namun tidak pernah digunakan. Misalnya penggunaannya hanya pada kalang if yang tidak pernah tercapai.

2.3 Garbage Collection

Dalam *computer science*, *garbage collection* adalah salah satu bentuk dari manajemen memori otomatis. *Garbage collector* atau biasa disebut *collector* bertugas untuk mengumpulkan memori yang sudah tidak dipakai lagi atau memori yang ditempati oleh objek namun tidak digunakan lagi. *Garbage collection* ditemukan oleh John McCarthy, seorang *computer scientist*, sekitar tahun 1959 sebagai abstraksi manajemen memori pada bahasa pemrograman LISP.

Garbage collection adalah kebalikan dari manajemen memori manual yang mengharuskan pembuat program menyatakan secara eksplisit alokasi dan dealokasi objek pada memori. Seperti memori manajemen lainnya, *garbage collection* dapat mengurangi waktu pemrosesan saat program berjalan, dan sebagai hasilnya dapat meningkatkan performansi secara signifikan.

III. Penerapan Breadth First Search pada *Garbage Collection*

Dalam penerapannya pada *garbage collection*, apabila diasumsikan struktur data dari memori adalah

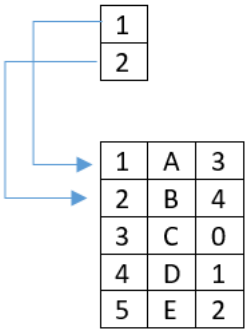
```
typedef struct tmemory{
        char value;
        tmemory *next[maxnext];
} memory;
```

di mana setiap memori memiliki nilai dan pointer yang menunjukkan alamat memori selanjutnya yang masih digunakan oleh program. Dengan menelusuri elemen-elemen yang ditunjuk oleh next, maka bisa ditentukan memori mana saja yang masih digunakan dan bisa membuang elemen yang tidak ditunjuk.

Tetapi karena memori tersebut sangatlah banyak dan belum menunjukkan memori mana saja yang menjadi akar dari graf memori tersebut, maka akar dari graf memori yang dianggap sebagai awal dari penelusuran juga perlu disimpan pada sebuah tabel, misalnya

```
typedef struct troot{
        memory* akar[maxroot];
} root;
```

dengan terdefinisinya akar dan memori, maka pencarian dapat mulai dilakukan. Struktur memori yang nantinya tercipta berdasar definisi tadi nantinya akan berbentuk seperti



gambar 2: struktur heap

Dari kedua tabel tersebut, tabel pertama adalah tabel akar yang menunjukkan elemen pertama graf. Selanjutnya tabel kedua adalah representasi memori dalam suatu program, yang secara berurutan kolomnya berarti alamat fisik, nilai, dan pointer ke elemen lain (elemen tetangga).

3.1 Implementasi BFS pada *Garbage Collector*

Dalam mengimplementasi BFS terdapat berbagai cara, cara yang paling sederhana adalah menyimpan sebuah antrian (queue) pada memori secara eksplisit. Cara ini mudah untuk diimplementasikan, dengan menyimpan dan menandai semua elemen yang telah dikunjungi. Nantinya semua elemen yang tidak ditandai didealokasi untuk bisa ditempati oleh objek lainnya.

Implementasi seperti di atas kurang baik karena jika hendak melakukan itu, memerlukan penanda untuk setiap bagian memori. Jika misalkan ada dua giga byte memori yang setiap misal empat bytenya diberi penanda sebesar satu byte, maka diperlukan sekitar 500 MB hanya untuk membuat penanda. Jika misalkan dibuat penanda untuk area lebih besar misal setiap 4 KB diberi penanda satu byte, hal tersebut memang bisa menghemat tempat secara signifikan, tetapi memori yang tidak digunakan menjadi lebih sulit untuk didealokasi karena dalam 4 KB tersebut haruslah semuanya kosong agar dapat didealokasi.

Salah satu cara untuk menghemat memori adalah dengan memindahkan data ke tempat awal atau tempat khusus pada memori dan menyimpan batas antara memori yang terpakai dan memori yang masih kosong. Apabila ingin mengalokasikan sejumlah memori, hanya cukup menambahkan pada batas dan mengincrement batas tersebut sejumlah memori yang dialokasikan.

3.1.1 *Garbage Collector* Menggunakan Antrian

Dalam menggunakan *garbage collection* menggunakan antrian, diperlukan suatu antrian Q yang menyimpan alamat pointer memori yang sedang dipindai. *Garbage collector* jenis ini akan membutuhkan dua *heap* karena memerlukan satu sebagai penampung memori asal sebelum *garbage collection* berjalan dan satu lagi sebagai penampung memori tujuan setelah *garbage collection* selesai.

Saat *garbage collection* dijalankan, terlebih dahulu akan membuat *heap* yang berisi ruang kosong dengan ukuran sama dengan total data yang akan dipindahkan.

```
void garbage_collection(memory
m,root r, queue q){
      int i=0;
      memory *temp[maxnext];
      int batas = minbatas;
      while (r.akar[i]!=NULL)
            q.add(r.akar[i++]);
      while (!(q.empty())){
            temp = q.get_child();
            batas = q.reallocate();
            q.del();
            q.add_child(temp);
      }
      m.del_after(batas);
}
```

Garbage collection di atas mengisi akar-akar terlebih dahulu pada antrian agar tidak ada yang terlewatkan. Setelah itu *garbage collector* akan melakukan pemrosesan untuk memindahkan data,

Dalam pemindahan data, selain nilai, bentuk graf juga harus diperhatikan, mengingat bentuk graf yang berbeda bisa menimbulkan arti yang berbeda. Misalkan dalam suatu kalang pada satu scope terdapat dua variabel yang dialokasikan, kedua variabel tersebut haruslah mengacu pada akar yang sama. Sehingga pada saat scope tersebut telah selesai, semua alamat memori yang ditunjuk baik secara eksplisit maupun implisit oleh akar dapat juga hilang.

Selain pemindahan memori, dalam alokasi ulang akar yang terdefinisi juga harus diperbarui karena akan mempengaruhi pada proses *garbage collection* yang berikutnya.

Setelah semua data berhasil di alokasi ulang, semua memori di luar batas akan dianggap bebas sehingga alokasi dapat dilakukan dengan menambahkan data setelah batas.

Proses *garbage collection* ini memerlukan lebih sedikit ruang dibandingkan memberi penanda. Hal ini dikarenakan hanya memori teralokasi saja yang akan dimasukkan ke dalam antrian.

3.1.2 *Garbage Collector* tanpa Antrian

Menggunakan antrian pada *garbage collector* terkadang masih memerlukan banyak memori dalam penyimpanan antrian, oleh karena itu dicarilah suatu cara agar bisa memanfaatkan BFS dalam *garbage collector* tanpa menggunakan antrian.

Tanpa menggunakan antrian, ruang penyimpanan yang dibutuhkan akan berkurang secara signifikan. Tetapi cara mengimplementasikannya pada *garbage collector* menjadi lebih sulit.

Seperti aplikasi *garbage collector* menggunakan antrian, *garbage collector* tanpa antrian juga memerlukan dua *heap*. dan akan membuang *heap* asal setelah proses pemindahan selesai. Dua *heap* yang digunakan biasa disebut *from-space* dan *to-space*. Proses pengubahan pointer p yang merujuk pada *from-space* menjadi sebuah pointer yang merujuk pada *to-space* disebut sebagai *forwading-pointer*. Proses *forwarding-pointer* kurang lebih seperti berikut,

```
memory* forward(memory* p){
   if (p.points(from_space)){
      if (p.f1.points(to_space))
         return p.f1;
      else{
         for(memory fi:p){
            fi.next=p.fi;
         }
         p.f1 = next;
         next=next + (sizeof(*p));
         return p.f1
      }
   }
   return p;
}
```

Fungsi di atas akan mengembalikan pointer hasil konversi dari suatu pointer p. Jika p sudah merujuk *to-space* akan mengembalikan p. Jika belum tetapi tetangganya sudah merujuk *to-space* kembalikan tetangganya, dan jika belum maka akan melakukan konversi dengan mengubahnya menjadi next. *Next* adalah alamat *to-space* yang masih kosong. Setelah konversi berhasil *next* akan bertambah sesuai dengan ukuran p.

Dalam melakukan *garbage collection*, program akan perlu menyimpan pointer yang sedang diperiksa dan elemen next.

```
void garbage_collection(root
akar){
   scan = to_space.begin();
   next = scan;
   for (memory* r:akar)
      r = r.forward();
   while scan < next{
      for (auto fi:*scan.next)
         scan.next=fi.forward();
      scan+=(sizeof (*scan));
   }
}
```

Proses *garbage collection* dimulai dengan melakukan *forward* pada setiap akar. Setelah itu melakukan forward pada setiap *field* dari pointer yang sedang discan. Misalkan suatu *root* berisikan

1
6
9

dan *from space* berisikan

1	51	6	e	3
2	2	0	i	4
3	54	0	d	5

4	4	0	g	0
5	5	0	a	0
6	52	8	b	7
7	7	10	k	0
8	55	0	c	0
9	53	0	j	10
10	10	0	f	0
11	11	0	h	12
12	12	11	l	0

Maka hal pertama yang dilakukan adalah melakukan forward pada root menjadi

51
52
53

dan *to space* akan terisi

51	51	6	e	3	scan
52	52	8	b	7	
53	53	0	j	10	
54					next

e, b, j adalah nilai dari memori yang ditunjuk oleh akar.

Karena semua akar telah termuat, yang perlu dilakukan hanya melakukan penyesuaian terhadap *to space*. Pada elemen pertama scan, 51 sudah menunjuk ke *to space*. Elemen keduanya, 6, menunjuk ke *from space*, tetapi 6 sudah termuat dalam *to space* sebagai 52. Oleh karena itu, 6 hanya perlu diubah menjadi 52. Elemen keempat, 3, menunjuk ke *from-space* yang belum termuat pada *to-space*, oleh karena itu, dilakukan *forwarding* pada 3 menjadi 54, sehingga tabel *to-space* menjadi

51	51	52	e	54	
52	52	8	b	7	scan
53	53	0	j	10	
54	54	0	d	5	
55					next

Proses ini dilakukan berulang-ulang sampai scan merujuk pada lokasi yang sama dengan next yang menunjukkan bahwa proses *garbage collection* telah selesai. *To-space* menjadi *heap* baru dan *from-space* dapat didealokasikan.

3.2 Implementasi DFS pada *Garbage collector*

Dalam implementasi DFS pada *garbage collector* lebih sulit dari pada implementasi BFS. Hal ini dikarenakan DFS lebih sulit untuk mempertahankan lokalitasnya.

3.2.1 Implementasi DFS menggunakan Tumpukan

Untuk mengimplementasikan DFS dengan tumpukan kurang lebih memiliki cara yang sama dengan BFS, hanya perlu mengganti proses untuk memasukkan pointer dalam tumpukan dan penghapusannya.

```
void garbage_collection(memory
m,root r, queue q){
   int i=0;
   memory *temp[maxnext];
   int batas = minbatas;
   while (r.akar[i]!=NULL)
      s.push(r.akar[i++]);
   while (!(s.empty())){
      temp = s.get_child();
      batas = s.reallocate();
      s.pop();
      s.push_child(temp);
   }
   m.del_after(batas);
}
```

Proses *garbage collection* juga kurang lebih sama, pertama-tama akar akan dimasukkan salam tumpukan s. Setelah itu elemen pertama tumpukan akan diambil tetangga-tetangganya. Jika elemen pertama sudah selesai di alokasi ulang, kemudian tetangga-tetangganya akan dimasukkan dalam tumpukan dan tetangga pertama akan di proses untuk di alokasi ulang.

Setelah semua alokasi ulang selesai, *garbage collection* akan melakukan pembebasan terhadap semua memori di luar batas.

3.2.2 Implementasi DFS tanpa Tumpukan

Implementasi DFS tanpa tumpukan juga mirip dengan BFS, hanya perlu mennganti dalam urutan *forwarding*, hal itu dapat dilakukan dengan proses rekursif menjadi

```
void garbage_collection(memory*
scan){
   for (int i=0;i<=maxnext;i++){
      if ((*scan).next[i] !=
NULL){
         scan.next[i] =
scan.next[i].forward();
         scan = scan + (sizeof
(*scan));

   garbage_collection(scan);
      }
   }
}
```

Perbedaan lain dalam implementasi ini dengan BFS adalah proses ini memerlukan pemrosesan akar dan inisialisasi scan dan next di luar prosedur utama *garbage collection*.

Program akan mengiterasi setiap elemen next dari memori, dan setiap elemen tersebut akan dilakukan proses *garbage collection*.

IV. KEKURANGAN

Kekurangan dari implementasi BFS dan DFS dalam *garbage collector* adalah perlunya melakukan penyalinan isi memori. Hal ini berdampak pada waktu *garbage collection*

berjalan saat memori yang kosong tidak mencukupi untuk proses penyalinan, *stack* harus dipindah ke atas dan membutuhkan waktu lebih lama dari seharusnya.

V. KESIMPULAN

Dalam performansinya, penggunaan BFS dan DFS tidak begitu jauh selisihnya dalam *garbage collector*. Tetapi iterasi DFS tidak cukup baik dalam mempertahankan lokalitas *heap* baru yang dihasilkan juga akan mempunyai lokalitas yang buruk untuk simpul-simpul yang saling bertetangga. Oleh karena itu performansi BFS akan lebih baik daripada DFS dalam *garbage collector*.

VI. UCAPAN TERIMA KASIH

Puji syukur penulis panjatkan kepada Tuhan YME karena atas izin-Nya makalah ini dapat selesai. Penulis mengucapkakan terima kasih kepada bapak Rinaldi Munir dan ibu Nur Ulfa Maulidevi atas bimbingannya selaku dosen strategi algoritma, serta kepada pendahulu-pendahulu yang telah memberikan karya-karya terkait algoritma penyelesaikan berbagai masalah terutama dalam masalah pencarian dan *garbage collection* sehingga membantu dalam menyelesaikan makalah ini.

REFERENSI

[1] https://lambda.uta.edu/cse5317/notes/node48.html, 5 Mei 2016 12.29 WIB.
[2] Slide Diktat Kuliah IF2211 Strategi Algoritma. Teknik Informatika, Institut Teknologi Bandung
[3] Tanenbaum, Andrew S. 2015. Modern Operating System.
[4] http://www.memorymanagement.org/glossary/g.html, 6 Mei 2016 19.00 WIB.
[5] https://lambda.uta.edu/cse5317/notes/node44.html, 6 Mei 2016 19.50 WIB.
[6] http://www.dynatrace.com/en/javabook/how-garbage-collection-works.html, 7 Mei 2016 18.50 WIB.
[7] http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html, 7 Mei 2016 19.48 WIB.
[8] http://useless-factor.blogspot.co.id/2008/02/quick-intro-to-garbage-collection.html, 8 Mei 2016 04.50 WIB.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 8 Mei 2016

ttd

Nugroho Satriyanto

13514038