2.7. Geflechte, die Halde und Freispeicher

Daten oder Objekte lassen sich durch Zeiger miteinander verflechten. Früher sprach man dann von "Geflechten", die im Speicher aufzubauen sind. Heute bezeichnet man diese Strukturen meist als Vernetzungen oder - mathematisch - als *Graphen* (vgl. das folgende Kapitel 3).

Um solche Geflechte oder Vernetzungen aufzubauen, muss in jedem Datenobjekt mindestens ein Zeiger existieren.

Existiert genau ein Zeiger, so kann man nur Listen aufbauen. Ab zwei Zeigern lassen sich stark vernetzte Strukturen realisieren.

Beispiel: Binäre Bäume. Siehe hierzu "Einführung in die Informatik I", WS 01/02, S. 149-152.

Grundsätzlicher Hinweis:

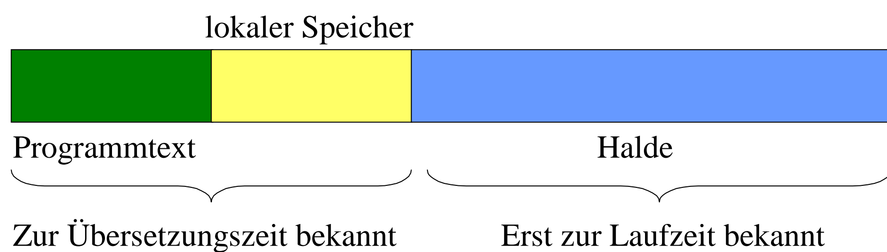
Wir identifizieren in diesem Kapitel 2 stets "Zeiger" und "Adresse eines Speicherplatzes".

Grund: Man beachte, dass heutige Rechner in der Regel einen ein-dimensionalen Speicher besitzen, auf dessen Speicherplätze über einen Index von 0 bis 2^s-1 (für eine natürliche Zahl s) zugegriffen wird. Wenn man Zeiger daher in einem solchen Rechner realisiert, so geschieht dies durch Angabe der Adresse derjenigen Speicherzelle, ab der das Objekt, auf das verwiesen wird, steht.

Wir haben in der Vorlesung bereits mehrfach angedeutet, wie man Programme im eindimensionalen Speicher des Rechners anlegt. Man unterteilt jedes Programm in drei Teile:

Zwei statische Teile (*Programmtext* und *lokaler Speicher*) für alle Bestandteile des Programms, deren Größe (Speicherplatzbedarf) während der Berechnung unverändert bleibt.

Dynamischer Teil (*Halde*) für alle Bestandteile, deren Größe oder Lage im Speicher sich ändern können.



Programmtext: Üblicherweise wird der Programmtext nach der Eingabe in einen Rechner nicht mehr verändert.

Dennoch findet eine Veränderung des Textes statt, sobald ein Unterprogramm aufgerufen wird. Die Bedeutung eines Unterprogrammaufrufs ist nämlich die "Kopierregel"; sie lautet: Ersetze den Aufruf durch den Rumpf des Unterprogramms; modifiziere den Rumpf abhängig von den Parametern, führe den modifizierten Rumpf aus, ersetze ihn am Ende wieder durch den Aufruf und fahre mit der nächsten Anweisung fort.

Es gibt jedoch eine Realisierung dieser Kopierregel, bei der der Programmtext nicht geändert wird, sondern bei der die korrekte Ausführung des Aufrufs durch einen Stack von Daten simuliert wird. In der Praxis wird daher der Programmtext während des Programmablaufs nicht geändert (vgl. Vorlesung Compilerbau).

Lokaler Speicher: Alle vom Programm benutzten Daten müssen letztlich über die Namen der Variablen (bzw. über die "Anker" von Listen und Geflechten) erreichbar sein. Genau diese Variablen, deren benötigter Speicherplatz zur Übersetzungszeit bekannt ist, werden in einem festen Speicherbereich, dem lokalen Speicher des Programms, abgelegt.

Felder mit festen Grenzen werden in der Regel ebenfalls hier gespeichert. Felder, deren Grenzen erst zur Laufzeit berechnet werden, kommen dagegen in die Halde (s.u.), jedoch wird für den Namen und die künftig einzutragende Größe und Lage jedes solchen Feldes Speicherplatz im lokalen Speicher reserviert, über den zur Laufzeit die Verbindung zum Feld hergestellt wird. Weiterhin müssen die Strukturen der deklarierten Datentypen im lokalen Speicher notiert werden (um die Zugriffe zu realisieren).

Halde (engl.: Heap): Teil 1: Dies ist ein Speicherbereich, dessen Größe hinreichend groß ist, um die dynamischen Felder und alle mittels **new** erzeugten Datenobjekte (Zugriff über Zeiger!) abzulegen. Wenn diese Datenobjekte im Laufe der Rechnungen nicht mehr gebraucht werden, sollten sie wieder frei gegeben (in Ada mit Hilfe des pragmas "Controlled" und der Prozedur FREE) und von anderen, neu erzeugten Datenobjekten genutzt werden. Die Verwaltung erfolgt auch über eine Freispeicherliste.

Werden die nicht mehr benötigten Speicherplätze nicht wieder frei gegeben, so liegen nach einiger Zeit in der Halde viele unnütze Datenobjekte herum (= Daten, auf die nicht mehr von irgendeinem lokalen Speicher aus zugegriffen werden kann). So kann die Halde rasch voll werden. Um dann weiterarbeiten zu können, müssen die nicht mehr benötigten Datenobjekte erkannt, ihre Speicherplätze frei gegeben und die Halde in geeigneter Weise umorganisiert werden (Speicherbereinigung).

Halde (engl.: Heap): Teil 2: Die Halde kann/muss ebenfalls zur Unterprogrammverwaltung dienen. Bei jedem solchen Aufruf muss für die lokalen Variablen des Unterprogramms Speicher zur Verfügung gestellt werden, der dynamisch mit weiteren (geschachtelten) Aufrufen wächst oder schrumpft.

Dieses Problem der beiden Speicher für dynamische Daten und für die Unterprogrammverwaltung kann man behandeln wie die Stackverwaltung für 2 Stacks: Von vorne nach hinten legt man in der Halde die dynamischen Daten ab, von hinten nach vorne lässt man die geschachtelten Aufrufe der Unterprogramme aller Programme (=Multistackverwaltung) wachsen.

Wir erläutern dies nun an einem Beispiel mit nur einem Programm.

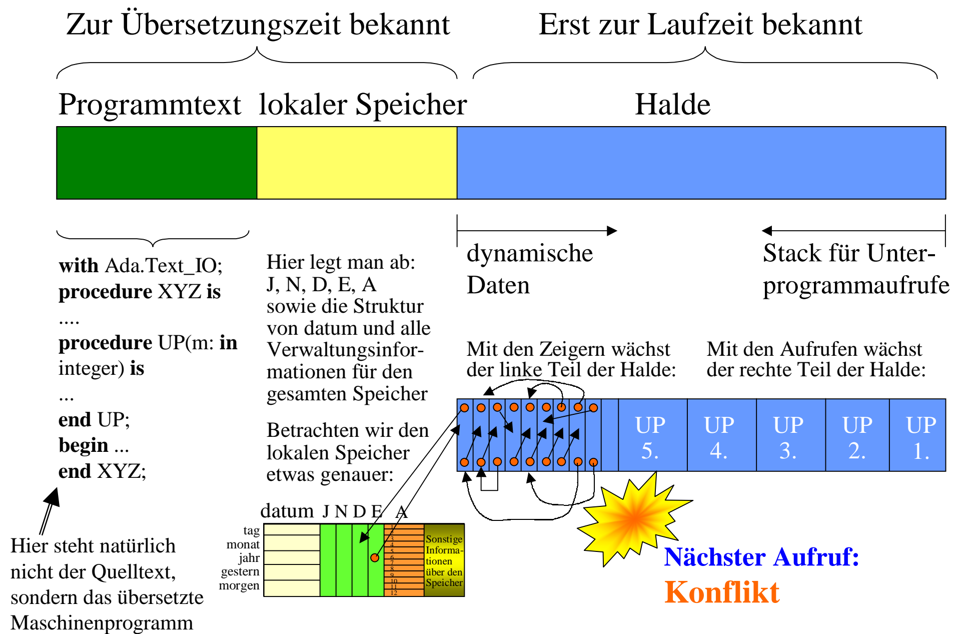

```
with Ada.Text_IO;

procedure XYZ is
type datum is
   record tag: 1..31; monat: 1..12; jahr: 1900..2100;
      gestern, morgen: access datum; end record;
J, N: integer; D: datum; E: access datum;
A: array (1..12) of character;
procedure UP (m: in integer) is
   B: array (1..m) of character;
   begin  ... UP(m+1); ...
   end UP;

begin  get(N); J := 2; D := (13, 5, 2002, null, null);
   E := D; E.gestern := D; E := new datum ...
   UP(N); ...
end XYZ;
```

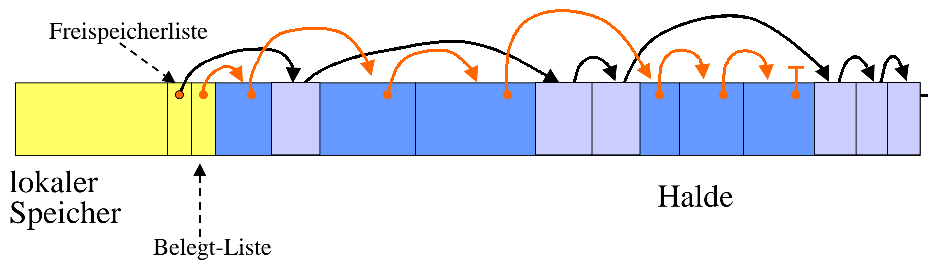
Lokaler Speicher

Halde



2.7.1 Freispeicherverwaltung

Die dynamischen Daten der Halde (linker Teil im vorigen Bild) sollen nun verwaltet werden. Hierzu tragen wir die freien Speicherplätze in eine "Freispeicherliste" ein, aber nicht jede Speicherzelle einzeln, sondern immer ganze "Datenblöcke". Diese können eine variable Größe besitzen. Skizze (hier *nur ein Programm*; prinzipiell nutzen viele Programme die Halde):



Benutzen mehrere Programme die Halde, so werden die "Freispeicherliste" und eventuell auch eine "Belegt-Liste" vom Betriebssystem verwaltet. Folgende Aufgaben sind unter anderen Fragestellungen zu lösen:

1. Ein Programm fordert einen Speicherplatzbereich der Größe "größe" an. Weise dem Programm einen geeigneten Bereich in der Halde zu und modifiziere die Freispeicherliste.

2. Ein Programm gibt einen Speicherplatzbereich wieder frei. Füge diesen Bereich "geschickt" in die Freispeicherliste ein.

3. Verschmelze aneinander grenzende freie Datenblöcke der Halde zu größeren Einheiten.

4. Falls keine Zuweisung erfolgen kann, ordne die Halde so um, dass alle freien Bereiche nebeneinander liegen (das ist nicht trivial). Füge hierbei alle Datenblöcke, die nicht mehr benutzt werden, in die Freispeicherliste ein (siehe 2.7.2).

5. Falls auch dies nicht erfolgreich ist, führe einen Austausch der Speicherinhalte mit dem Hintergrundspeicher durch (Stichwort: Seitenaustauschstrategien, Paging; siehe Vorlesungen im Bereich "Betriebssysteme").

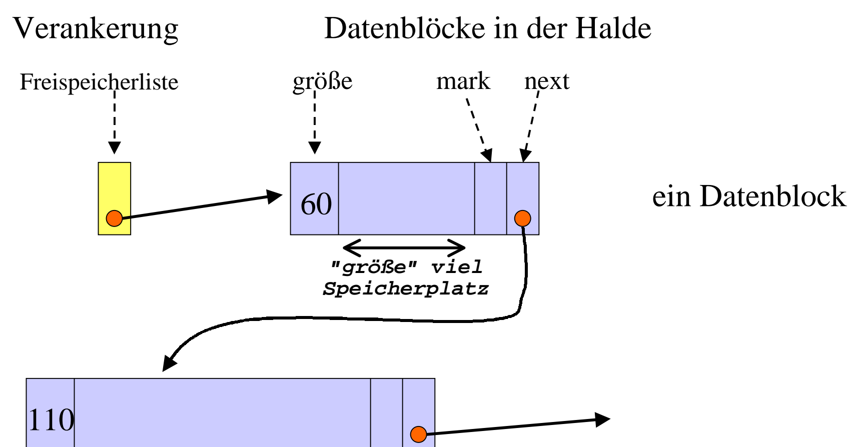
Um diese Aufgaben durchzuführen, muss die Freispeicherliste oft durchlaufen werden, wobei wir die Datenblöcke, die zur Freispeicherliste gehören, markieren, um sie später "erkennen" zu können. Ein Datenblock muss also neben dem Inhalt, den das jeweilige Programm hineinschreibt, mindestens seine Größe, ein Markierungsfeld und den Verweis auf den nächsten freien Datenblock enthalten.

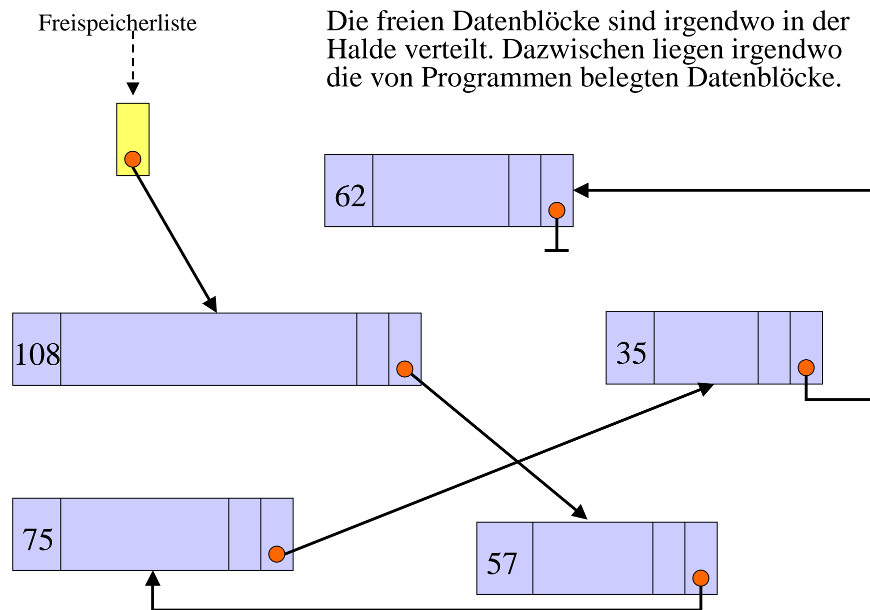
Wir legen daher folgenden Datentyp "Block" für Datenblöcke fest. Es sei eine natürliche Zahl "maxgröße" (= die maximale Größe an Speicherplätzen je Datenblock) vorgegeben:

```
type Block;
type Blockzeiger is access Block;
type Block is record
        größe: 1..maxgröße;
        ...  < Komponenten, die insgesamt genau "größe" viele
                Speicherplätze belegen >   ...
        mark: Boolean;
        next: Blockzeiger;
        end record;
```

Jeder Block belegt also größe + x viele Speicherplätze in der Halde, wobei x die Zahl der Speicherplätze für "größe", "mark" und "next" bezeichnet. (generic-Formulierung in Ada? Selbst!)

Skizze:





<u>1. Aufgabenstellung</u>: Ein Programm fordert einen Datenblock mit m Speicherplätzen an.

<u>Algorithmus 1</u>: **First Fit**

Gehe die Freispeicherliste durch, bis ein Datenblock D mit größe ≥ m gefunden ist.
Mache hieraus zwei Datenblöcke: Einen mit m+x und einen mit größe-m-x Speicherplätzen (x = Speicherplatz für größe, mark und next, siehe Datentyp Block auf Folie Seite 88).
Füge diese beiden Datenblöcke in die Freispeicherliste anstelle des Blocks D ein.
Klinke den ersten dieser beiden Datenblöcke aus der Freispeicherliste aus und weise ihn dem Programm zu.
Hinweise: Falls der zweite Block "zu klein" ist, vermeide die Aufspaltung in zwei Datenblöcke und weise D dem Programm zu. Falls kein Block D existiert, rufe die Speicherbereinigung auf, vgl. Abschnitt 2.7.2.

Algorithmus 2: **Best Fit**
Gehe die gesamte Freispeicherliste durch und ermittele den kleinsten Datenblock D mit größe ≥ m.
Fahre anschließend fort wie bei "First Fit".

Welche Strategie ist besser?

Bei beiden Methoden entstehen im Lauf der Zeit viele kleine Datenblöcke, die verstreut in der Halde liegen. Diese sog. "Fragmentierung" des Speichers erfordert häufige Aufrufe der Speicherbereinigung. In der Praxis erweist sich die Best-Fit-Strategie gegenüber der "First-Fit-Strategie" als schlechter, da hierbei besonders kleine Datenblöcke entstehen; außerdem muss bei Best-Fit stets die gesamte Freispeicherliste durchlaufen werden.

Aus der Praxis weiß man: Solange der freie Speicher etwa ein Drittel der Halde ausmacht, ist die First-Fit-Strategie gut anwendbar. Wird aber der freie Platz geringer, so muss oft eine zeitaufwändige Speicherbereinigung durchgeführt werden, die zu Wartezeiten bei den "Kunden" führt.

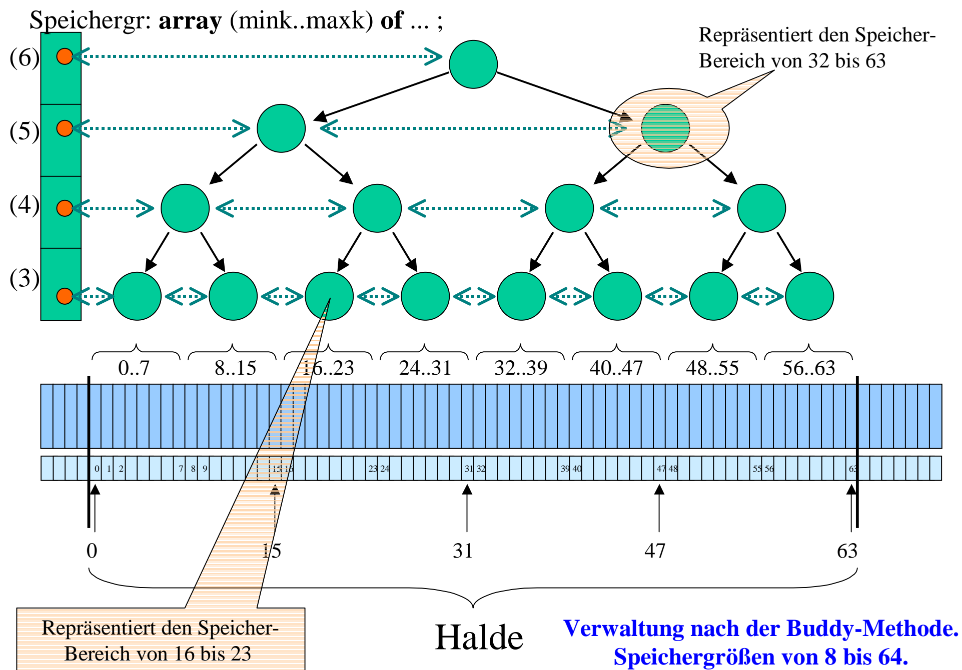
Recht nachteilig ist auch die Zeit, die beim Durchlaufen der Liste verstreicht. Zwei Ideen zur Verbesserung:

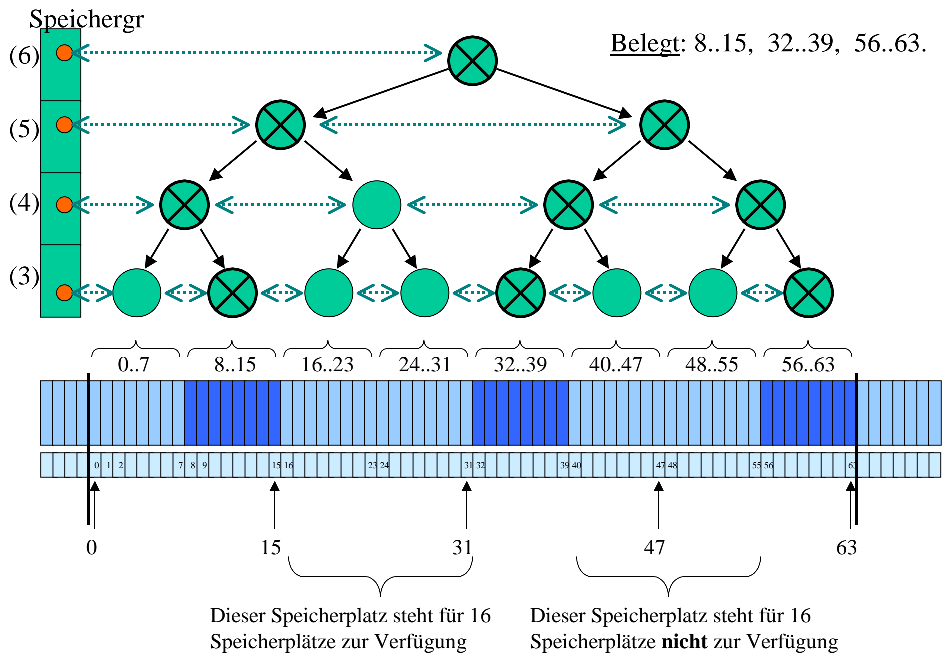
- Halte die Freispeicherliste stets nach der Größe der Datenblöcke sortiert. Nachteil: Das Einfügen freigegebener Speicherbereiche ist dann aufwändig.
- Lege einen binären Suchbaum über die Freispeicherliste. Nachteil: Dies kostet relativ viel Speicherplatz. Da solche Suchbäume selbst wieder dynamische Datenstrukturen sind, sollte dieser Platz in der Halde bereit gestellt werden.

Lassen sich die Vor- und Nachteile gegeneinander auswägen? Wir stellen kurz einen alten Vorschlag vor, der die Situation einigermaßen verbessert, aber oft einige Nachteile beibehält, die sog. Buddy-Methode. Buddy = (engl.) Kamerad, Kumpel.

Idee: Das Verfahren legt einen gleichverzweigten binären Baum über den Speicher (also über die Halde). Das Aufspalten und das Verschmelzen sind aber nicht beliebig möglich.

Vorgehen: Die Halde wird in Datenblöcke unterteilt, deren Länge jeweils eine Zweierpotenz ist. Jeder Datenblock muss genau 2^k Speicherplätze belegen für eine natürliche Zahl k mit **`mink`** $\leq k \leq$ **`maxk`**. (Man schränkt in der Praxis k ein z.B. zwischen **`mink`** = 9 und **`maxk`** = 20.) Jedem Datenblock wird genau einer seiner beiden Nachbarn als "Buddy" zugeordnet.





Wir nummerieren die Halde also von 0 bis $2^{maxk} - 1$ durch. Die kleinste Blockgröße sei 2^{mink}. Alle Speicherblöcke mit festem `mink` ≤ k ≤ `maxk` stehen in einer Liste, erreichbar über den Zeiger des Feldelements Speichergr(k).

Zu jedem Speicherblock, der an der Adresse x beginnt und die Größe 2^k besitzt, sei **buddy$_k$(x)** die Anfangsadresse seines Buddy (dieser liegt entweder links oder rechts von ihm und besitzt die gleiche Größe).

Es gilt:

$$\mathbf{buddy_k(x)} = \begin{cases} x + 2^k \text{, falls } x = 0 \mod 2^{k+1} \\ x - 2^k \text{, falls } x = 2^k \mod 2^{k+1} \end{cases}$$

Wir programmieren die Freispeicherverwaltung nicht aus, sondern geben nur die Vorgehensweisen an.

Anfangs werden alle Speicherbereiche als frei markiert.

Speicheranforderung: Ein Programm fordert einen Datenblock der Größe m an. Es sei $2^{k-1} < m \leq 2^k$.
Die Speicherverwaltung durchläuft dann die Liste, die über Speichergr(k) erreichbar ist.
Wird hier ein freier Speicherbereich gefunden, so wird er dem Programm zugewiesen; zugleich werden dieser Bereich, alle Knoten in seinem Unterbaum und seine Vorgänger im Baum bis zur Wurzel als belegt markiert.
Wird kein freier Speicherbereich gefunden, so lege die Speicher-anforderung in einer Warteschlange des Systems ab, sende dem Programm einen "Wartehinweis" und prüfe später erneut.

Beispiel: Wird in der Situation der Folie auf Seite 96 ein Bereich der Größe 14 angefordert, so wird ab Speichergr(4) die Liste der Speicherblöcke der Größe $2^4 = 16$ durchsucht. Bereits der zweite Block ist frei, so dass der Bereich 16..31 zugewiesen wird.

Speicherfreigabe: Ein Programm gibt einen Datenblock der Größe 2^k wieder frei. Dieser Block wird in der Liste zu Speichergr(k) als frei markiert. Ist sein Buddy frei, so wiederhole diesen Vorgang mit seinem Vorgängerknoten.

Beispiel: Wird in der Situation der Folie auf Seite 96 der Bereich 8..15 der Größe 8 freigegeben, so kann dieser Block, aber auch sein Vorgänger und dessen Vorgänger frei gegeben werden, so dass anschließend die linken drei als belegt markierten Knoten im Baum wieder als frei markiert sind.

Vorteile der Buddy-Methode: Einfach zu handhaben und das Verfahren bewährt sich in der Praxis hinreichend gut.
Nachteile: Benachbarte Bereiche, die nicht Buddys sind, können nicht verschmolzen werden, und es entsteht eine interne Fragmentierung, da immer nur Blöcke von der Länge einer Zweierpotenz zugewiesen werden können.

2.7.2 Speicherbereinigung (garbage collection)

Wir nehmen nun an, ein Programm schreibt die Halde mit dynamischen Datenstrukturen, und zwar mit verzeigerten Strukturen, voll. Es ist absehbar, dass in Kürze kein Speicherplatz mehr zur Verfügung steht.

Nun muss geprüft werden, ob die Datenobjekte, die in der Halde stehen, wirklich alle benötigt werden oder ob man sie löschen und auf diese Weise neuen Speicherplatz bereitstellen kann.

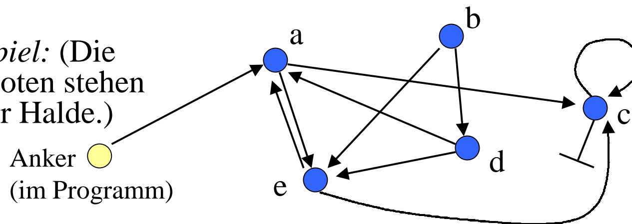
Hierfür verfolgt man alle Zeiger, die vom lokalen Speicher des Programms ausgehen und markiert alle Datenobjekte, die auf diese Weise erreichbar sind. Die nicht-markierten kann man löschen.

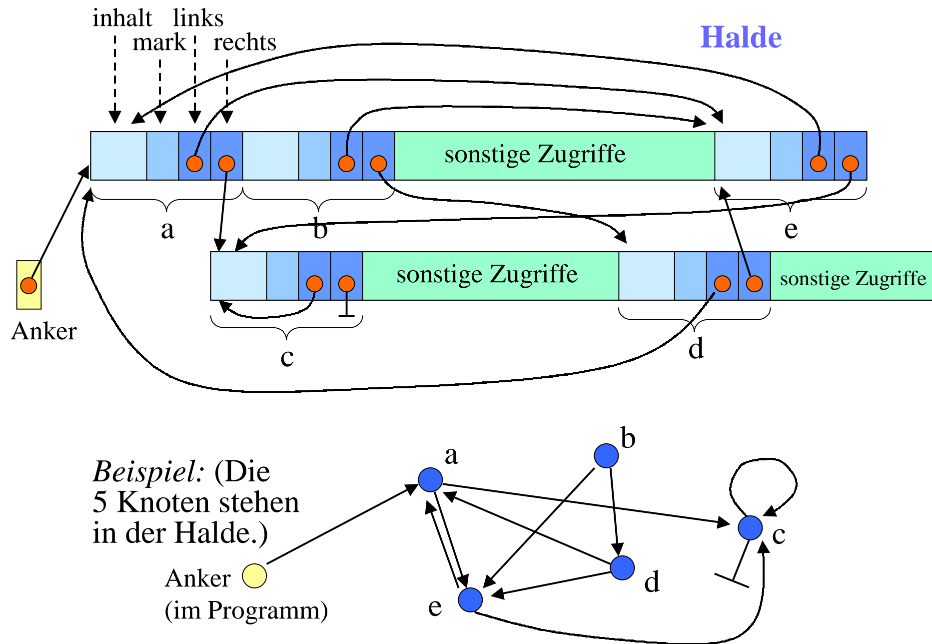
Um das Vorgehen zu erläutern, genügt es, Datenobjekte mit zwei Zeigern zu betrachten, also Objekte des Typs

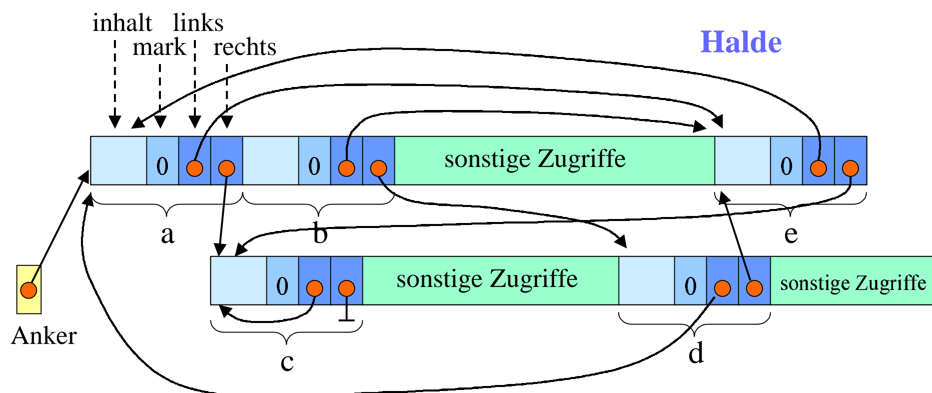
type Knoten;
type Kante **is access** Knoten;
type Knoten **is record**
    inhalt: ...
    mark: Boolean;
    links, rechts: Kante;
end record;

Die Knoten b und d sind vom Programm aus nicht erreichbar.

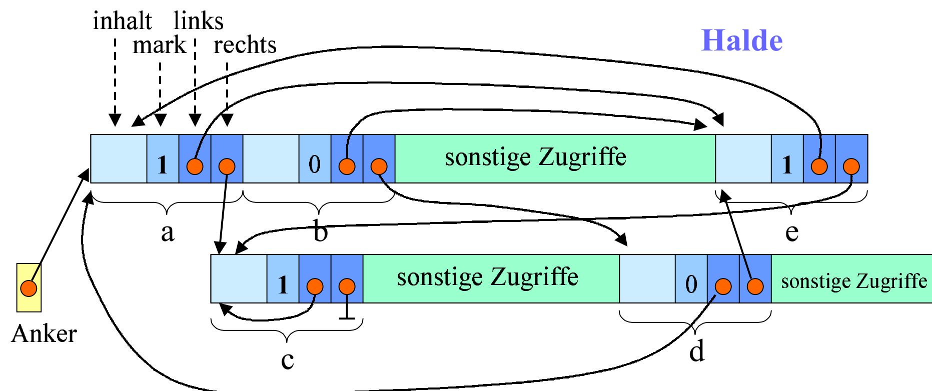
Beispiel: (Die 5 Knoten stehen in der Halde.)







Ziel muss es nun sein, die Knoten b und d als löschbare Knoten zu erkennen. Hierzu durchläuft man ausgehend von "Anker" alle Zeiger und markiert die hierbei erreichten Knoten mit true (oder einer "1"). Dies geschieht für alle "Anker", die aus dem lokalen Speicher in die Halde verweisen. Dann löscht man alle mit "0" markierten Datenobjekte und schiebt den Speicher geeignet zusammen.



Ergebnis des Durchlaufs: ausgehend von "Anker" wurden alle Zeiger nachverfolgt und die hierbei erreichten Knoten mit einer "1" und die nicht erreichbaren mit einer "0" markiert.

Anschließend eventuell den Speicherbereich neu organisieren.

Hinweis: Wenn man weiß, dass die Zeigerstrukturen keine Kreise bilden (also "azyklisch" sind), dann kann man in jeden Knoten einen "Verweiszähler" aufnehmen, der angibt, wie oft auf dieses Objekt verwiesen wird. Wird ein Knoten mit zwei Zeigern hinzugefügt, so müssen die Verweiszähler der Knoten, auf die diese Zeiger zeigen, jeweils um 1 erhöht werden. Wird ein Knoten gelöscht, so muss man die Verweiszähler in den beiden Objekten, auf die die Zeiger links und rechts zeigten, um jeweils 1 erniedrigen. Wird ein Verweiszähler hierbei 0, dann muss man auch in allen nachfolgenden Knoten den Verweiszähler um 1 erniedrigen. Bei dieser Methode kann man zu einem gegebenen Zeitpunkt genau alle die Knoten löschen, deren Verweiszähler 0 ist.

Bei zyklischen Strukturen funktioniert dieses einfache Verfahren aber nicht mehr. (Selbst durchdenken.)

Wir kommen nun zum
Algorithmus zur Markierung der erreichbaren und der unerreichbaren Knoten:

Schritt 1:
Markiere alle Knoten in der Halde mit "false" (bzw. mit 0).

Schritt 2:
Markiere alle Knoten in der Halde, die vom lokalen Speicher direkt erreicht werden können, mit "true" (bzw. mit 1).

Schritt 3 (eigentlicher Algorithmus): Die Halde möge von Adresse 0 bis Adresse M im Speicher nummeriert sein. Jeder Knoten möge genau r Speicherplätze (Adressen) belegen.
u, v sind vom Typ Knoten, i und j sind Adressen in der Halde.


```
i := 0;                    -- i ist die Adresse des betrachteten Knotens
while i <= M loop
   j := i + r;             -- j wird die Adresse des nächsten Knotens
   if  der Knoten mit Adresse i besitzt mindestens einen
          Nachfolger (d.h.: (links /= null) or (rechts /= null))
       and  dieser Knoten ist mit "true" markiert
   then  if  (links /= null) and (der Knoten u, auf den links
                verweist, ist mit "false" markiert) then
             markiere den Knoten u mit "true";
             j := Minimum (j, Adresse von u)  end if;
         if  (rechts /= null) and (der Knoten v, auf den rechts
                verweist, ist mit "false" markiert) then
             markiere den Knoten v mit "true";
             j := Minimum (j, Adresse von v)  end if;
   end if;
   i := j;                 -- zum nächsten Knoten gehen
end loop;
```

Idee dieses Verfahrens: Durchlaufe die Knoten von vorne nach hinten in der Halde. Es interessieren nur die mit true markierten Knoten (nur sie sind bisher vom Programm aus erreichbar). Betrachte deren beide Nachfolgeknoten. Markiere sie mit true und setze das Verfahren an der minimalen Adresse der drei Knoten
- nächster Knoten in der Halde
- linker Nachfolgeknoten
- rechter Nachfolgeknoten

fort.

Auf diese Weise gelangt man schließlich an alle erreichbaren Knoten.

Aufwand dieses Verfahrens?

Im ungünstigsten Fall beim Durchlauf durch die Halde ist die **if**-Bedingung erst beim letzten Knoten erfüllt und dessen Verweis führt auf den ersten Knoten zurück.

Nach dem zweiten Durchlauf geschieht das Gleiche mit dem vorletzten Knoten usw.

Wenn n die Zahl der Knoten in der Halde ist, so würde man also $n + (n-1) + (n-2) + ... + 2 + 1 = n \cdot (n+1)/2 = O(n^2)$ Schritte ausführen müssen. Wegen $n \approx M/r$ erhält man also ein $O(M^2)$-Verfahren mit konstanter Speicherkomplexität (M = Anzahl der Speicherplätze in der Halde).

In der Tat erweist sich dieser Algorithmus in der Praxis auch im Mittel als ein quadratisch mit M wachsendes Verfahren.

Hinweis:
Es ist klar, wie man dieses Verfahren auf Knoten, die mehr als zwei Nachfolger haben können oder deren Größe im Datenobjekt selbst gespeichert ist, erweitern kann:
- **for all** Nachfolgeknoten (im äußersten **then**-Teil),
- ersetze j := i+r durch j:=i+größe_des_aktuellen_Knotens.

Das oben genannte Verfahren eignet sich besonders dann, wenn man (fast) keinen freien Speicherplatz mehr zur Verfügung hat. Gibt es dagegen noch Speicherplatz, den man für einen Stack S nutzen kann, dann empfiehlt sich folgender deutlich schnellerer Algorithmus, der die weiter zu verfolgenden Zeiger im Stack S ablegt:

Schritt 1: wie oben.
Schritt 2: Markiere alle Knoten in der Halde, die vom lokalen Speicher direkt erreicht werden können, mit "true" und lege sie (bzw. ihre Adressen) im Stack S ab.
Schritt 3:

```
while not isempty(S) loop
    while not isempty(S) and (top(S) hat keinen Nachfolger)
        loop pop(S) end loop;
    if not isempty(S) then
        K := top(S);  pop(S);
        if (K.links /= null) and then (not K.links.mark) then
            K.links.mark := true;  push(S,K.links);  end if;
        if (K.rechts /= null) and then (not K.rechts.mark) then
            K.rechts.mark := true;  push(S,K.rechts);  end if;
    end if;
end loop;
```

Aufwand dieses Stack-Verfahrens?

Das Verfahren durchläuft jeden Zeiger, der in einem Knoten auftritt, höchstens einmal. Da es höchstens doppelt so viele Zeiger wie Knoten gibt, handelt es sich bei Schritt 3 also um ein $O(n')$-Verfahren (n' = Zahl der erreichbaren Knoten in der Halde).

Allerdings bezahlt man diese Schnelligkeit mit dem benötigten Speicherplatz für den Stack S. Dieser kann bis zu $n/2$ Knoten groß werden. Im Mittel wird man aber deutlich weniger Platz brauchen.

Die uniforme Zeitkomplexität dieses Stack-Verfahren wird also vor allem durch Schritt 1 bestimmt, welcher M/r Zeiteinheiten benötigt. Insgesamt ergibt sich damit ein $O(M)$-Verfahren sowohl bzgl. der Zeit als auch bzgl. des Platzes.

Man kann nun die beiden Algorithmen kombinieren:

Solange noch genügend Platz für den Stack S vorhanden ist, arbeite nach dem Stackverfahren. Sobald der Stack aber überläuft, schalte auf das andere Verfahren um, bis wieder Platz für den Stack da ist.

Man erkennt nun auch die Abhängigkeit der Speicherbereinigung von der jeweiligen Programmiersprache: Man muss wissen, wie die Datenobjekte / Blöcke / Knoten usw. aufgebaut sind, um Zeiger auch als Zeiger erkennen zu können. Andererseits kann natürlich das Betriebssystem ein universelles Datenformat vorgeben, in dem zum Beispiel die Informationen über Zeiger an vorgegebenen Stellen notiert werden müssen.

Nachdem wir nun die erreichbaren Knoten bzw. Datenblöcke mit "true" markiert haben, kann man alle diese in die Freispeicherliste (oder die Buddy-Verwaltung) eintragen und normal weitermachen.

Oft möchte man jedoch den Speicher "zusammenschieben" ("kompaktifizieren") und dabei auch die im Laufe der Rechnungen entstandenen kleinen Fragmente (nicht nutzbaren Speicherbereiche) beseitigen. Dieser Kompaktifizierungs-Algorithmus ist bei beliebiger Verzeigerung einigermaßen aufwändig.

Diese und weitere Fragen zur Verwaltung von Programmen und Daten lernen Sie in Vorlesungen über Betriebssysteme oder auch in speziellen Praktika.

2.8. Zusammenfassung

Behandelte Datenstrukturen (und korrespondierende Kontrollstrukturen):
Feld (array; Vektoren, Matrizen, ...),
Verbund (record; kartesisches Produkt),
Vereinigung (varianter record, union),
Potenzmengen (set of ...)
Folgenbildung (Listen, Zeigertypen; freies Monoid)
Geflechte (Graphen, siehe Kap. 3)

Konkrete Verfahren:
n-dimensionale Felder auf eindimensionale abbilden:
Speicherabbildungsfunktion.
n Stacks möglichst gut verwalten (u.a.: Garwick-Algorithmus)
Freispeicherverwaltung
Speicherbereinigung (garbage collection, ohne Kompaktifizierung)

Historische Hinweise:

Mit der Entwicklung der ersten Computer in den 1940er Jahren entstanden auch sogleich eindimensionale Felder, da diese genau die Speicherstruktur wiedergaben. Allgemeine Felder finden sich bereits in den Programmiersprachen der 1950er Jahre (Fortran 58, Algol 60, APL). Verbunde und Folgen von Buchstaben treten in Cobol auf (ab 1961). Das gesamte Konzept der Datenstrukturen wurde komplett in Algol 68 (Standard: 1975) zusammengeführt. Dessen "gut verständlicher" Anteil wurde von Nikolaus Wirth in die Sprache PASCAL (1972) eingebracht, die bis heute als "didaktisches Vorbild" für Programmiersprachen gilt.

Mit den ersten Compilern Anfang der 1960er Jahre wurden Speicherabbildungsfunktionen und Optimierungen bei <u>for</u>-Schleifen eingeführt.
Dass sehr allgemeine Datenstrukturen korrekt übersetzbar sind, demonstrierten die Compiler der Sprache SIMULA 67 (ab 1965) und etwas später von PL/1. Probleme bereiteten aber die ganz allgemeinen Datenstrukturen von Algol 68, bei denen kartesische Produkte, Vereinigungen, Potenzmengen, Funktionenbildung usw. beliebig miteinander verknüpft werden können: Die Laufzeitsysteme wurden derart kompliziert, dass jeder Algol-68-Compiler gewisse Einschränkungen machen musste.
Mitte der 1960er Jahre entstanden die ersten Betriebssysteme, die mehrere Programme gleichzeitig verwalten konnten. Ab dieser Zeit entwickelte man diverse Verfahren für die Speicherverwaltung (wie Multistack, Freispeicher, Bereinigung usw.).