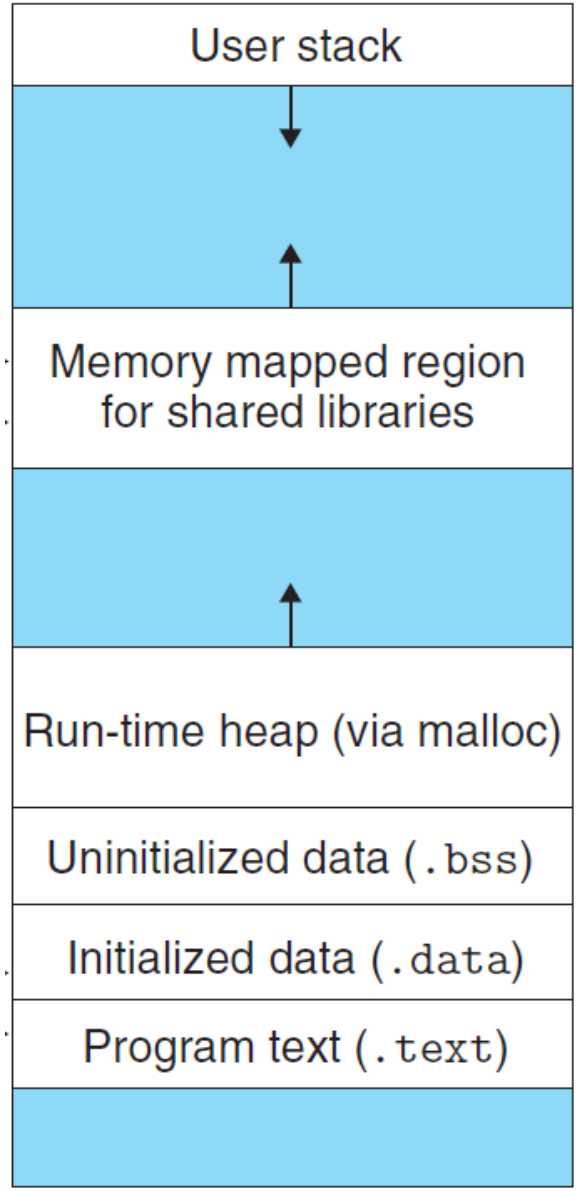
# תכנות מתקדם
# מצגת 7

## הקצאה דינמית

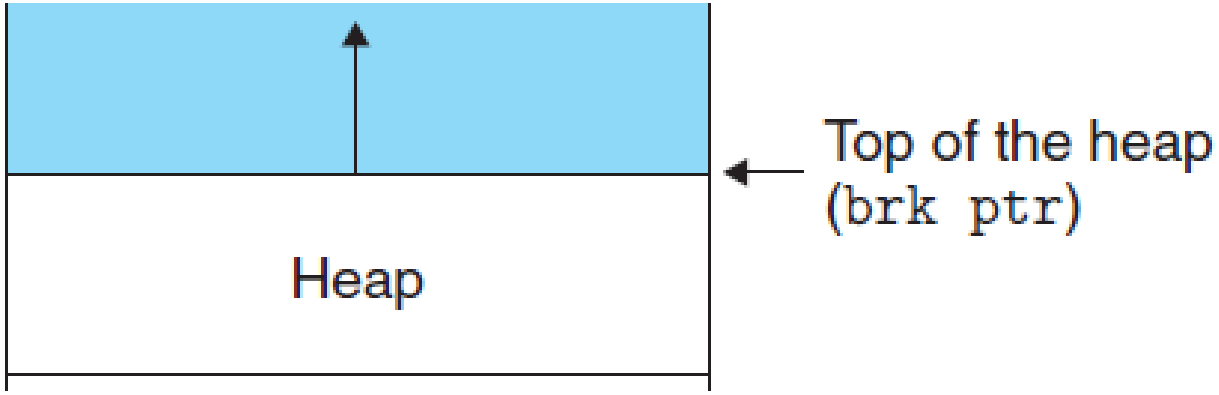
# זיכרון דינמי




- כשתכנית נטענת לזיכרון היא מחולקת לאזורים:
  - קוד, נתונים, ערימה, מחסנית וספריות.
- הקצאת **הקוד והנתונים הגלובליים** היא **סטטית**.
  - גודל ומקום האובייקטים בהקצאה **סטטית** ידוע בזמן קומפילציה ואינו משתנה במהלך התכנית.
- הקצאת **הערימה והמחסנית** היא **דינמית**.
  - גודל ומקום האובייקטים בהקצאה **דינמית** אינו ידוע בזמן קומפילציה ומשתנה במהלך התכנית.
  - הערימה והמחסנית גדלים בכיוונים הפוכים כדי לנצל היטב את מרווח הזיכרון שביניהם.
  - **הערימה** משמשת להקצאת אובייקטים בזמן ריצה.
  - **המחסנית** משמשת לניהול קריאות לפונקציה.

# הקצאת זיכרון בערימה

- תכנית תקצה אובייקטים בערימה:
  - אם גודל האובייקט לא ידוע מראש.
  - כדי ליצור שיתוף של נתונים עבור כמה פונקציות באמצעות מצביעים.
- כדי להקצות ולשחרר אובייקטים בערימה, התכנית קוראת באופן ישיר או עקיף לפונקציות ספרייה שמנהלות את הערימה, בדרך כלל malloc.
  - malloc מנהל את הערימה כאוסף של בלוקים בגדלים שונים שחלקם תפוסים וחלקם פנויים.
- במערכת ההפעלה ישנו מצביע לראש הערימה בשם brk.
  - אם לא נשאר מקום פנוי בערימה, malloc יבקש ממערכת ההפעלה להעלות את המצביע brk ובכך להגדיל את הערימה.



# new -ו malloc

- בשפת C (ו- C++) אפשר להקצות ולשחרר זיכרון עם **פונקציות** הספרייה `malloc()` ו- `free()`:

```
MyStruct *p = (MyStruct*) malloc(sizeof(MyStruct));
```

malloc מחזיר מצביע סתמי (void) לתחילת בלוק זיכרון בגודל שבקשנו.
הכתובת המוחזרת מכוונת (aligned) לכפולה של 8 או 16.

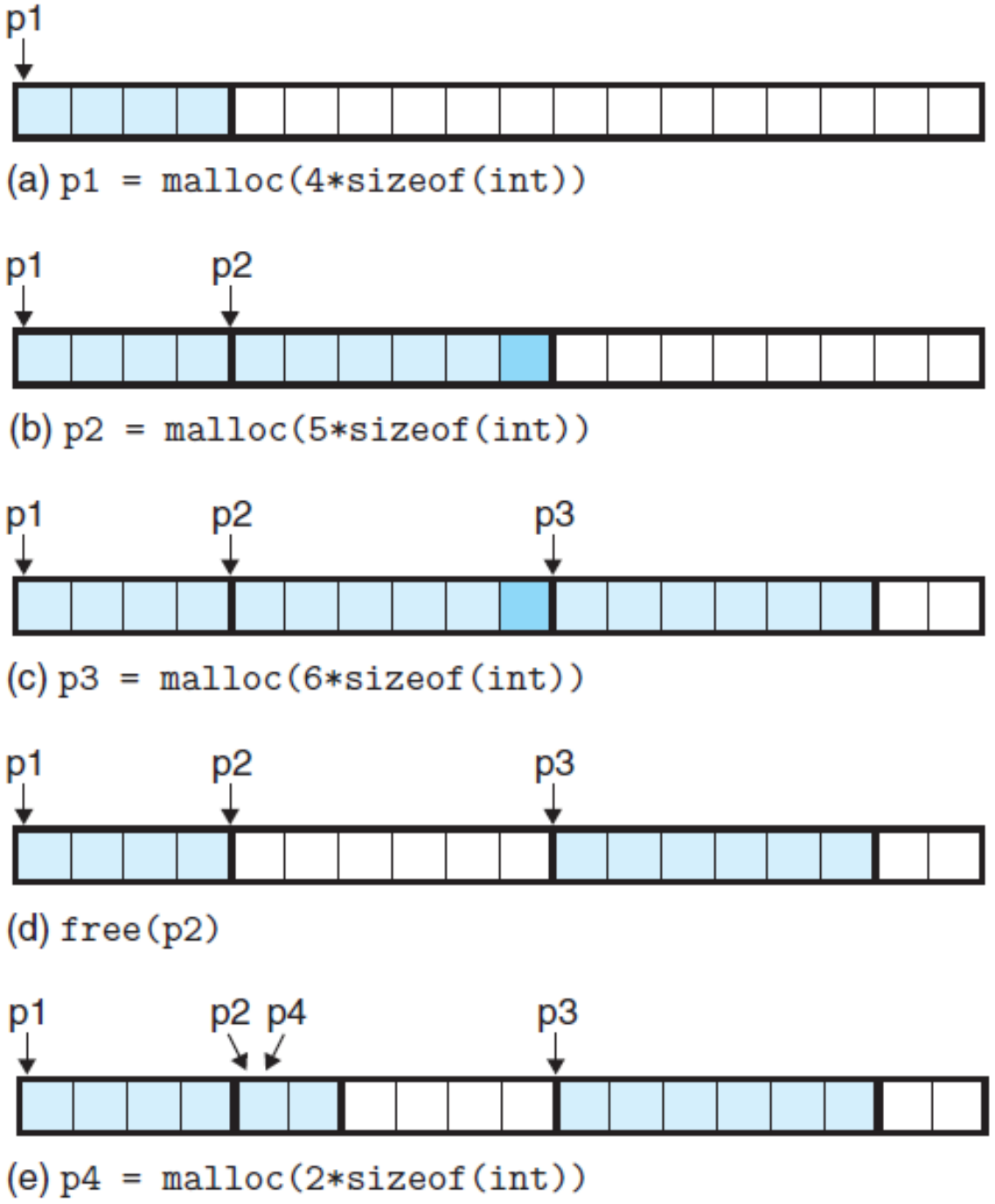
- בשפת C++ בדרך כלל נשתמש **באופרטורים** `new` ו- `delete`.

```
MyClass *fp = new MyClass(1,2); // object initialized
```

- `new` הוא **אופרטור** שבנוסף להקצאת הזיכרון (באמצעות `malloc()`) גם מפעיל את הבנאי (בדוגמה, מעביר לו `1,2` כארגומנט)
- `delete` הוא **אופרטור** שמשחרר זיכרון (באמצעות `free()`) וגם מפעיל את המפרק.

# הקצאת זיכרון על ידי malloc

- בדוגמה, הערימה מכילה מילים בגודל ארבעה בתים, כל מילה מיוצגת על ידי משבצת.
- הקריאות ל- ()malloc מחזירות מצביע לכתובת מכוונת לכפולה של שתי מילים (שמונה בתים).
- אחרי הקריאה ל- ()free, p2 ממשיך להצביע על הבלוק ששוחרר.

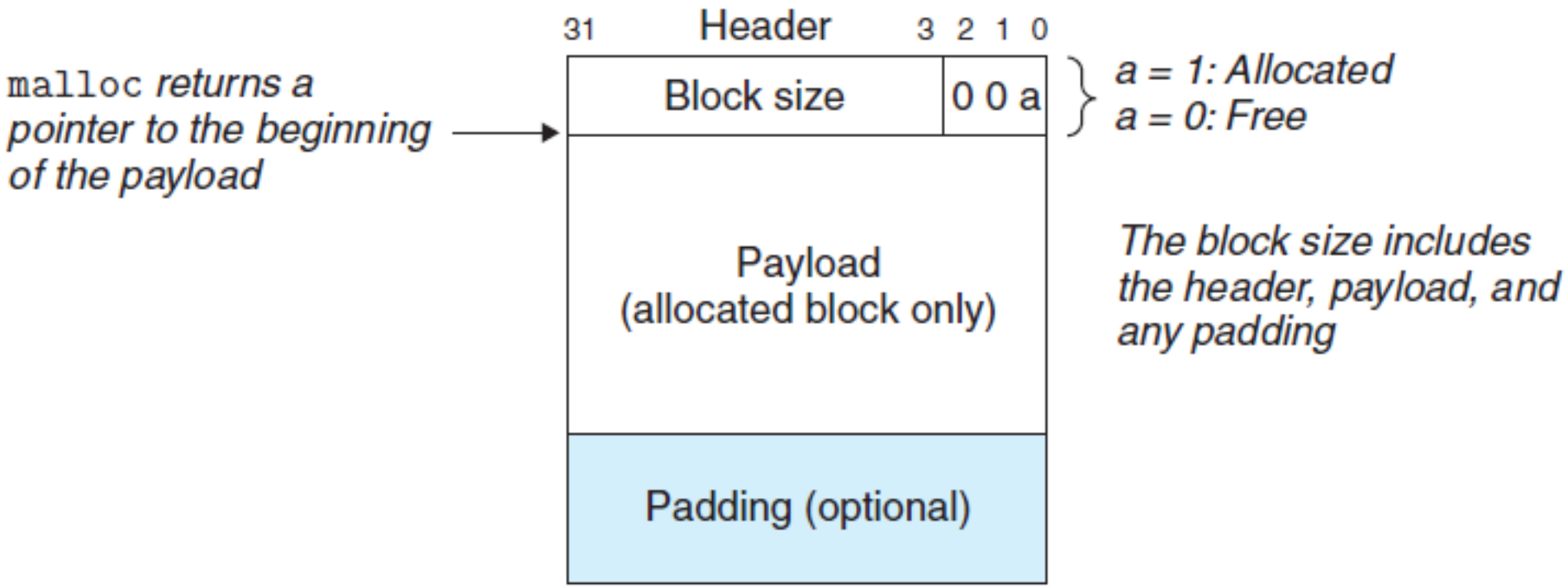


# בעיות שצריך לפתור במימוש malloc

- תגובה מהירה לבקשת ההקצאה.
  - ניהול רשימת הבלוקים הפנויים באופן שיאפשר חיפוש מהיר של בלוק פנוי.
- הקצאות שיאפשרו ניצול יעיל של הזיכרון וימנעו בזבוז:
  - שבירה פנימית נגרמת כאשר גודל הזיכרון המוקצה בבלוק גדול מהגודל המבוקש - בגלל הקצאה בכפולות של שמונה או בגלל גודל מינימלי של בלוק.
  - שבירה חיצונית נגרמת כאשר יש מספיק זיכרון פנוי אבל הוא מחולק לחלקים קטנים שאף אחד מהם לא מתאים לגודל המבוקש.

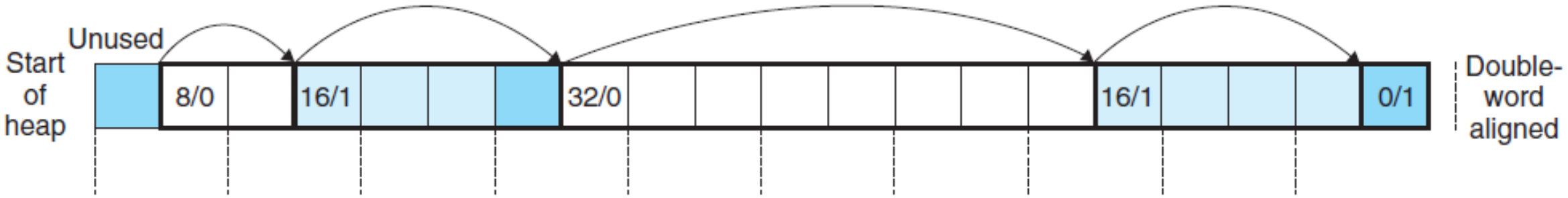
# הקצאת בלוק

- כש- malloc מקצה בלוק עבור תכנית שרצה, הבלוק מכיל בנוסף לזיכרון המוחזר לתכנית גם **כותרת**.
- הכותרת מכילה את גודל הבלוק וביט שמסמן אם הבלוק תפוס או פנוי.
- מאחר שההקצאה היא בכפולות של שמונה בתים, שלושת הביטים הימניים של הגודל תמיד יהיו 0 ולכן אפשר להשתמש בהם לצורך סימון.



# ניהול רשימת הבלוקים הפנויים - ללא מצביעים

- כדי להקצות זיכרון, malloc צריך למצוא בלוק פנוי בערימה בגודל מתאים.
- לצורך זה הוא צריך רשימה של הבלוקים הפנויים כך שיוכל לעבור על הרשימה ולבחור מתוכה בלוק מתאים.
- מאחר שכל בלוק מכיל גודל, זה יוצר רשימה ללא מצביעים (implicit), ולכן malloc יוכל לעבור מבלוק לבלוק ולבדוק את הסימון אם פנוי ואת הגודל אם מתאים.



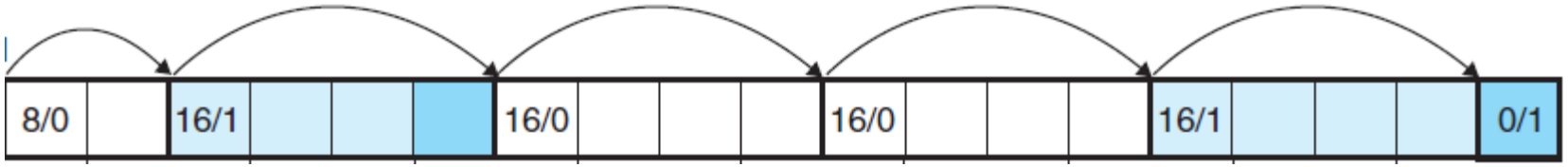

בציור, הבלוקים הפנויים בצבע לבן, זוג המספרים הוא גודל וסימון אם תפוס

# איזה מהבלוקים הפנויים יוקצה

- כאשר תכנית מבקשת הקצאה בגודל מסוים, הזיכרון המוקצה צריך להיות בגודל שווה או גדול מהגודל המבוקש.
- אם ברשימת הפנויים ישנם כמה בלוקים שהם מספיק גדולים מי מהם יוקצה ?
- **First fit** מקצה את הבלוק הראשון ברשימת הבלוקים שהוא מספיק גדול.
  - יתרון: מותיר בלוקים גדולים בסוף הרשימה.
  - חיסרון: יוצר בלוקים קטנים בתחילת הרשימה.
- **Next fit** מתחיל כל חיפוש מהמקום שהחיפוש הקודם הסתיים.
  - יתרון: סיכוי טוב שימצא יותר מהר בלוק מתאים מאשר **First fit.**
  - חיסרון: לא מותיר בלוקים גדולים בסוף הרשימה.
- **Best fit** מקצה את הבלוק הקטן ביותר שמתאים.
  - יתרון: ניצול טוב של הזיכרון.
  - חיסרון: זמן חיפוש ארוך.

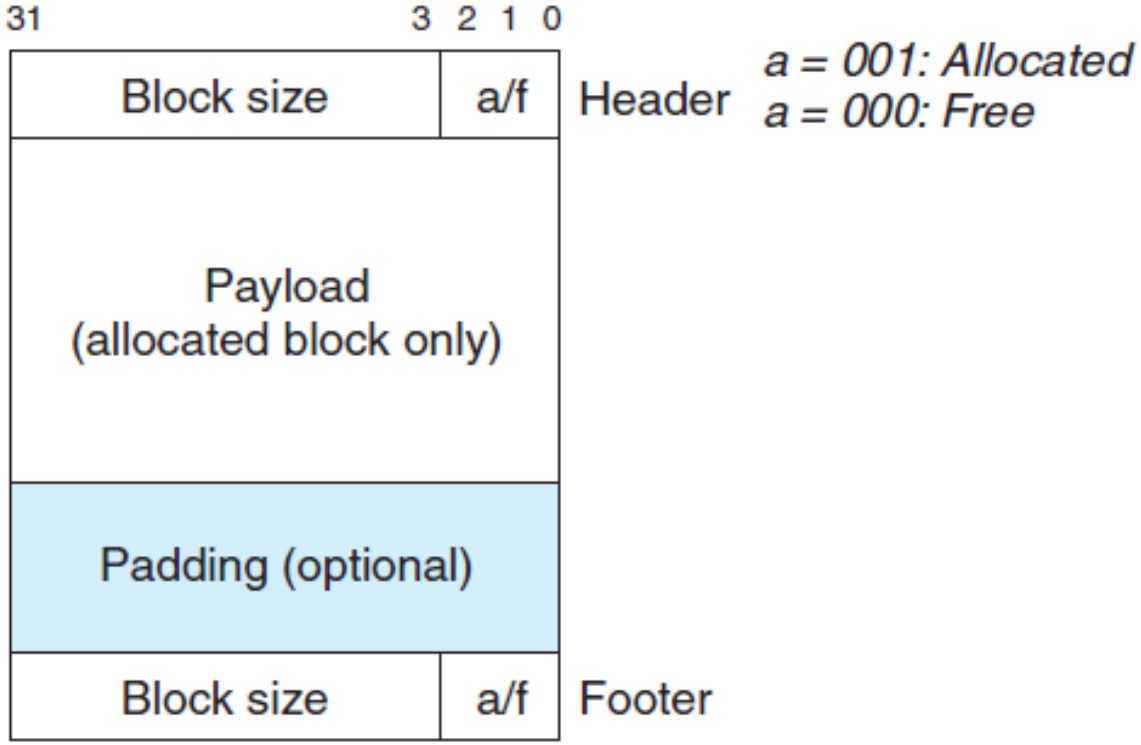
# פיצול ואיחוד של בלוקים

- **בהקצאה**, אם הבלוק שנבחר גדול מהגודל המבוקש, malloc **יפצל** את הבלוק לשני בלוקים אחד לצורך הקצאה והשני יישאר פנוי - כדי למנוע **שבירה פנימית**.
- **בשחרור** (free), יתכן שלפני או אחרי הבלוק המשוחרר ישנו בלוק פנוי, free **יאחד** את הבלוקים - כדי למנוע **שבירה חיצונית**.
  - איחוד עם **הבלוק הבא** הוא מהיר, פונקציית השחרור free קיבלה מצביע לבלוק הנוכחי ולכן היא יודעת את גודלו.
  - לפי הגודל היא יכולה להגיע לבלוק הבא ולבדוק אם הוא פנוי.
  - אם הוא פנוי, כדי לאחד אותו עם הבלוק הנוכחי צריך להוסיף את הגודל שלו לגודל של הבלוק הנוכחי, האיחוד אם כן מתבצע **בזמן קבוע**.
  - כדי לבדוק אם **הבלוק הקודם** פנוי יש צורך לעבור על הרשימה מהתחלה.
  - יוצא שכל שחרור (free) יתבצע **בזמן לינארי** לפי גודל רשימת הבלוקים.

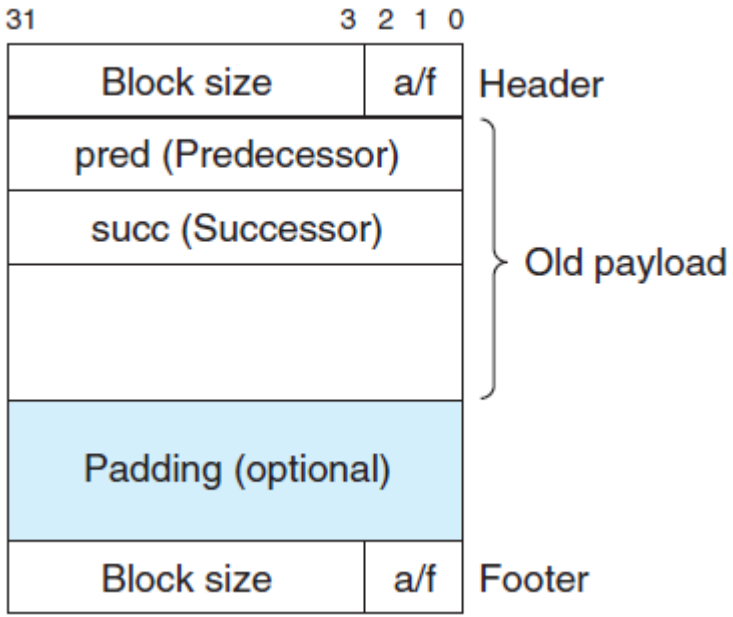


# איחוד בזמן קבוע באמצעות תגים

- אם נעתיק את הכותרת לסוף הבלוק נוכל לבצע גם **איחוד לאחור בזמן קבוע**.
- הכותרת שהעתקנו נמצאת במרחק של מילה לפני הכותרת של הבלוק הבא.
  - כאשר המשחרר (free) מקבל מצביע לבלוק, הוא יוכל בזמן קבוע למצוא את העתק הכותרת של הבלוק הקודם לבדוק אם הבלוק הקודם פנוי ולדעת את גודלו.



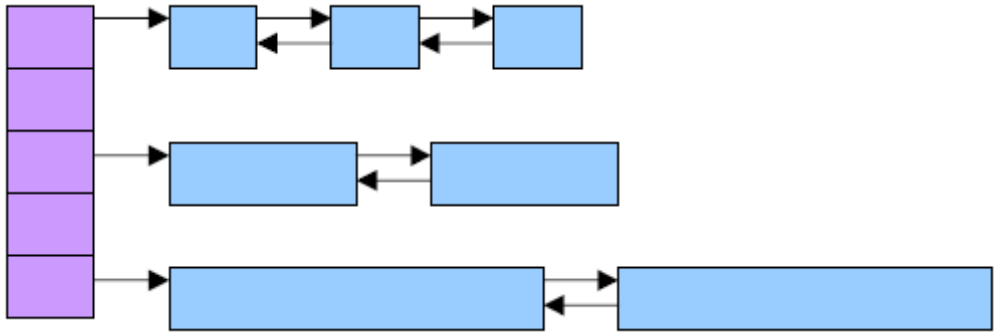
# רשימת הבלוקים הפנויים - עם מצביעים




- כשרשימת הבלוקים הפנויים היא ללא מצביעם, זמן ההקצאה של בלוק הוא **לינארי** לפי מספר **הבלוקים בערימה**.
- אפשר לשפר את **זמן ההקצאה** אם ניצור רשימה של **הבלוקים הפנויים** על ידי הוספת מצביעים לכל **בלוק פנוי**.
  - המצביעים לא תופסים מקום כאשר הבלוק מוקצה.
  - רשימה זו אינה לפי הסדר בזיכרון.
- זמן ההקצאה עם רשימת הבלוקים הפנויים הוא כעת **לינארי** לפי מספר **הבלוקים הפנויים**.
- **זמן השחרור נשאר קבוע**
  - נכניס בלוק שהשתחרר לראש רשימת הבלוקים הפנויים.
  - זמן **האיחוד** הוא כאמור **קבוע** אם משתמשים בתגים.

# ניהול הבלוקים הפנויים באמצעות כמה רשימות

- עם רשימה אחת של בלוקים פנויים, צריך לחפש בכל הבלוקים הפנויים.
- אפשר להקטין את זמן החיפוש, על ידי שימוש בכמה רשימות של בלוקים פנויים.
  - כל רשימה מכילה בלוקים בגודל שווה.
  - הרשימות מכילות בלוקים בגדלים שונים, לדוגמה: 8 בתים, 16 בתים, 32 בתים, ...
- ההקצאה תהיה מרשימת הבלוקים הקטנים ביותר שמכילים את הגודל המבוקש.
- אם הגודל המבוקש קטן מהבלוק, הוא לא יפוצל.
- בלוקים סמוכים לא יאוחדו.
- זמן ההקצאה והשחרור קבועים:
  מקצים את הבלוק שבראש הרשימה וכשבלוק משתחרר מחזירים אותו לראש הרשימה.
- חיסרון:
- שבירה פנימית – הקצאה שמשתמשת בחלק מהבלוק.
- שבירה חיצונית – אם יש צורך בהקצאה גדולה,
  אי אפשר לקחת מהבלוקים הקטנים.



# בעיות בניהול ידני של זיכרון דינמי

- **דליפת זיכרון - memory leak**
- התכנית מקצה זיכרון, לא משחררת, ומאבדת את המצביע.
- ללא מצביע אין אפשרות להשתמש בזיכרון או לשחרר אותו.
- זה לא מפריע לריצת התכנית כל עוד יש מספיק זיכרון.

```
void leak1() {
    Object *x = new Object;
    return;
}
void leak2() {
   Object *x = new Object;
   x = new Object;
}
```

valgrind

# בעיות בניהול ידני של זיכרון דינמי

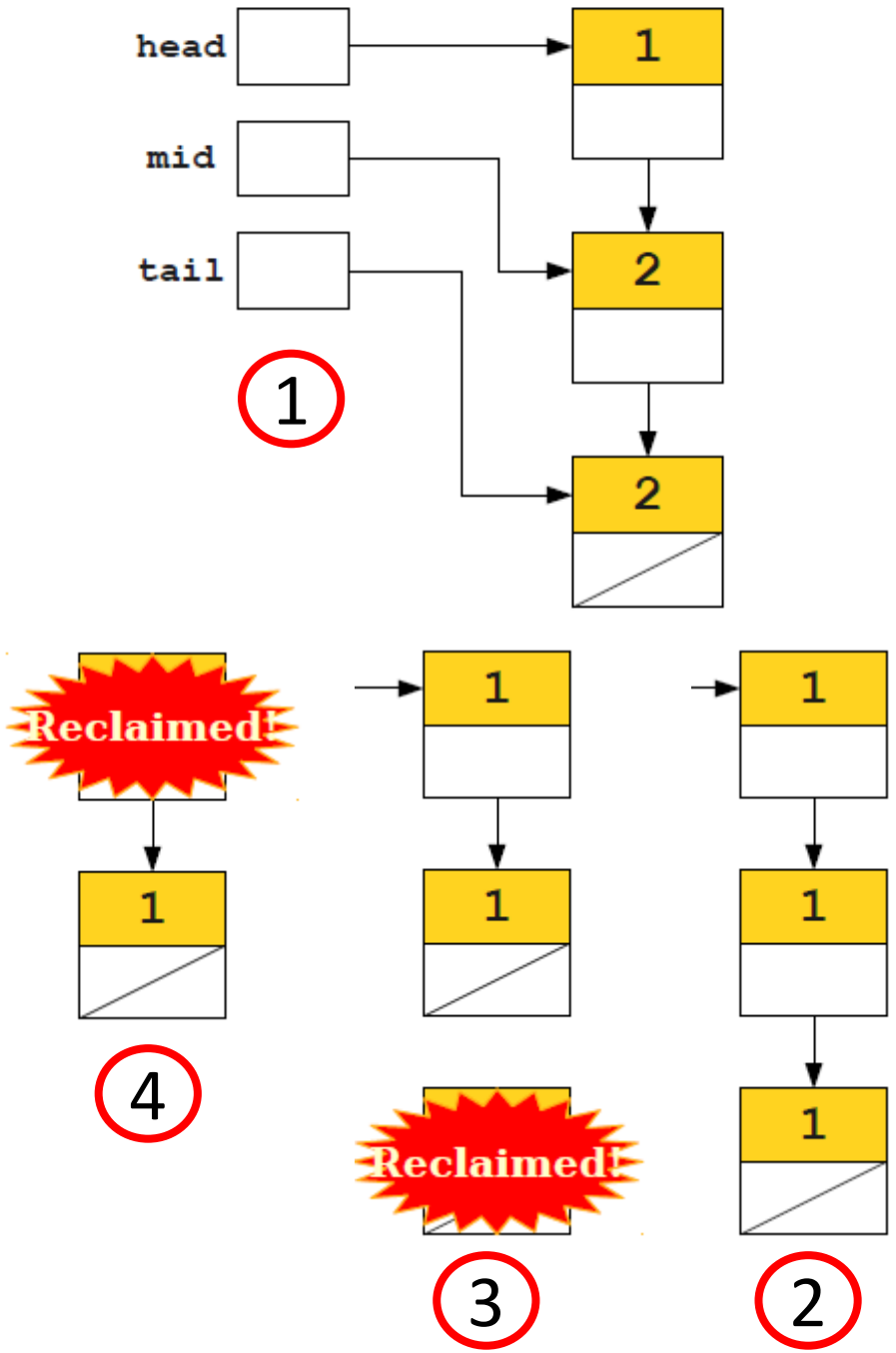
- **מצביע משוחרר -** `dangling pointer`
- התכנית מקצה זיכרון, משחררת, וממשיכה להשתמש במצביע כאילו לא שוחרר.
- שימוש במצביע משוחרר יגרום בדרך כלל לתוצאות שגויות.
  - קריאה מהמצביע עלולה להחזיר תוכן אקראי.
  - כתיבה למצביע עלולה לשנות תוכן של אובייקט אחר.
  - שחרור המצביע המשוחרר (`double free`) עלולה לשחרר אובייקט אחר.

```
for (p = head; p != NULL; p = p->next) { // Error
    free(p);
}
for (p = head; p != NULL; p = q) {       // Correct
    q = p->next; free(p);
}
```

# ניהול אוטומטי של זיכרון דינמי באמצעות Reference Counting

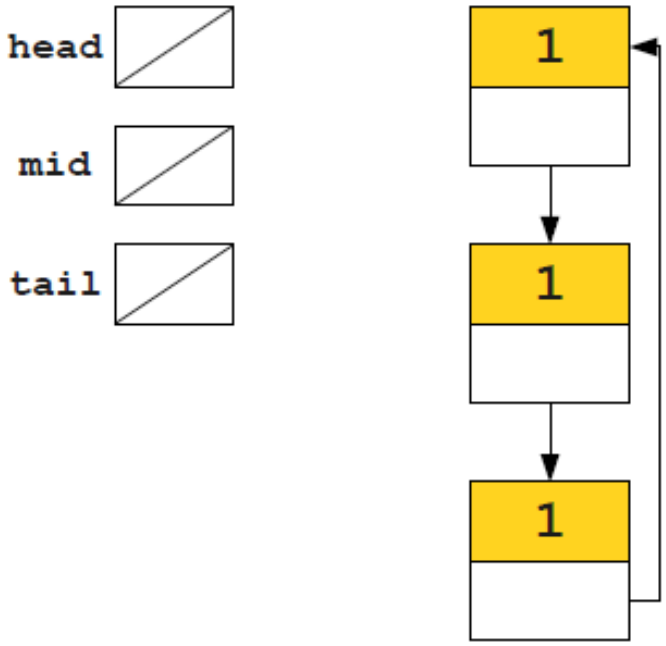
- לכל אובייקט שאליו יש מצביע יהיה **מונה** (`reference counter`) ובו נספור כמה מצביעים יש לאותו אובייקט.
- בכל פעם שנוסיף מצביע לאובייקט **נוסיף** אחד למונה:
  - כשתכנית יוצרת מצביע לאובייקט הערך ההתחלתי של המונה הוא **אחד**.
  - כשמעבירים מצביע כארגומנט לפונקציה **נוסיף** אחד למונה.
- בכל פעם שתכנית מסירה מצביע לאובייקט **נפחית** אחד מהמונה:
  - כאשר פונקציה שמכילה מצביע מקומי מסתיימת **נפחית** אחד מהמונה.
  - בהשמה של מצביע, **נוסיף** לאובייקט של צד ימין **ונפחית** מאובייקט של צד שמאל.
- אם המונה הגיע לאפס - **משחררים** את הזיכרון של אותו אובייקט.
  - אם האובייקט המשוחרר מכיל מצביעים לאובייקטים אחרים, מפחיתים את המונה של אותם אובייקטים.
  - זה יכול לגרום לשחרור של אותם אובייקטים וכן הלאה.

# דוגמה ל- Reference Counting




```
class LinkedList {
      ...; LinkedList next;
}
void f() {
LinkedList *head = new LinkedList;
LinkedList *mid = new LinkedList;
LinkedList *tail = new LinkedList;
head->next = mid;
mid->next = tail; 1
mid = tail = null; 2
head->next->next = null; 3
head = null; 4   // all reclaimed
}
```

# Reference Cycles - בעיה

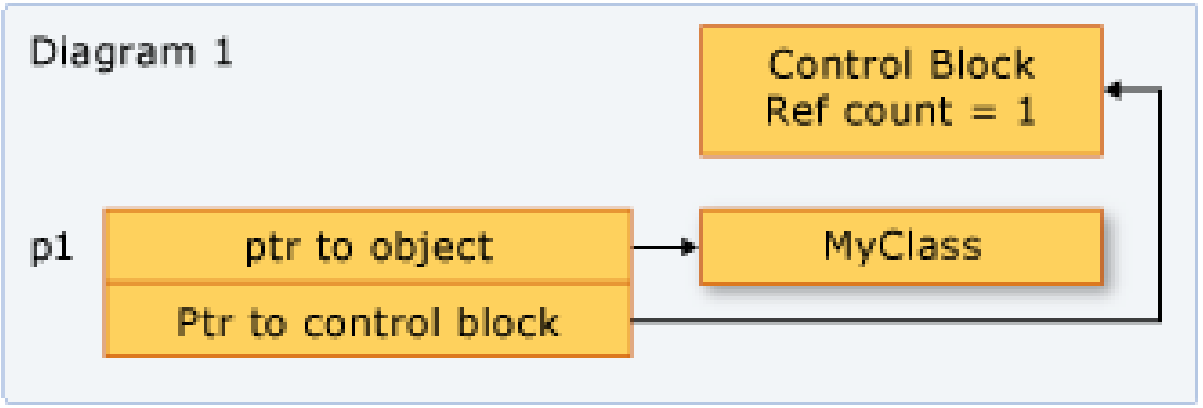


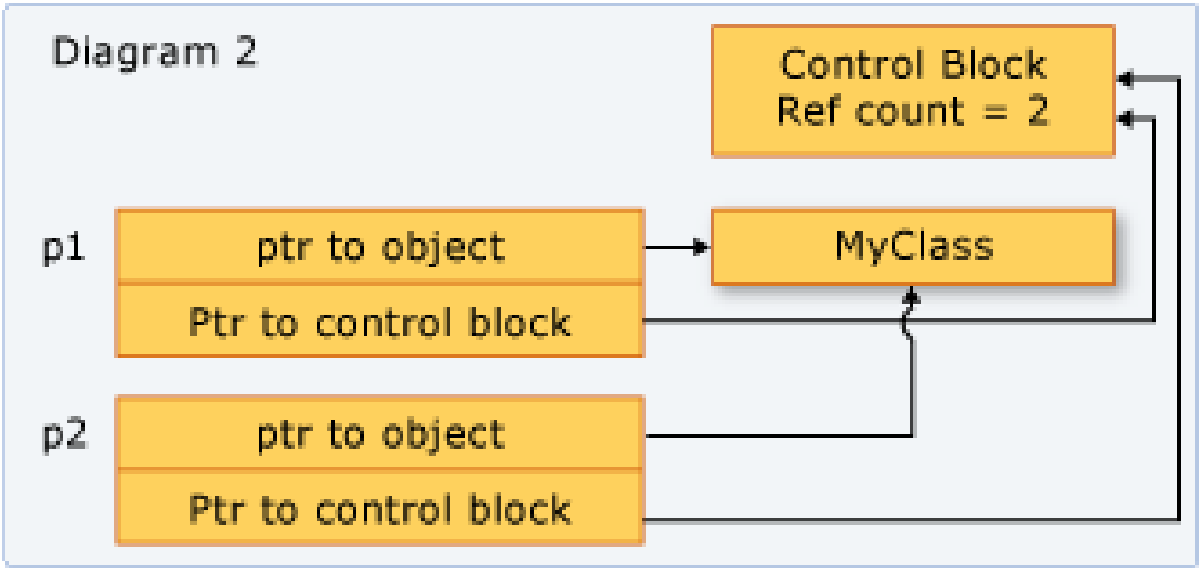

- **reference cycle** הוא קבוצה של אובייקטים שמצביעים זה לזה בצורה מעגלית.
- מאחר שלכל אובייקט יש מצביע שמצביע עליו, המונה שלו (**reference counter**) אינו מתאפס ולא נשחרר אותו.
- זאת למרות שאין מצביע שמצביע לקבוצה והיא לא נגישה (**unreachable**).

```
LinkedList *head = new LinkedList;
LinkedList *mid = new LinkedList;
LinkedList *tail = new LinkedList;
head->next = mid;
mid->next = tail;
tail->next = head;
head = mid = tail = null;
```

# shared_ptr - שיחרור אוטומטי ב- C++ עם reference counting

- `shared_ptr` היא מחלקה בספריה הסטנדרטית שמממשת מצביע עם **מונה**.
- `shared_ptr` מכיל את המצביע לאובייקט אליו מצביעים וכן מצביע למונה שסופר את מספר המצביעים לאובייקט.
- **הבנאי** של `shared_ptr` **מעלה** את המונה באחד.
- **המפרק** של `shared_ptr` **מפחית** מהמונה אחד ואם הגיע לאפס **משחרר**.
- המחלקה `shared_ptr` מגדירה את האופרטורים `*` ו- `->` כך שאובייקט של המחלקה מתנהג כמו מצביע.



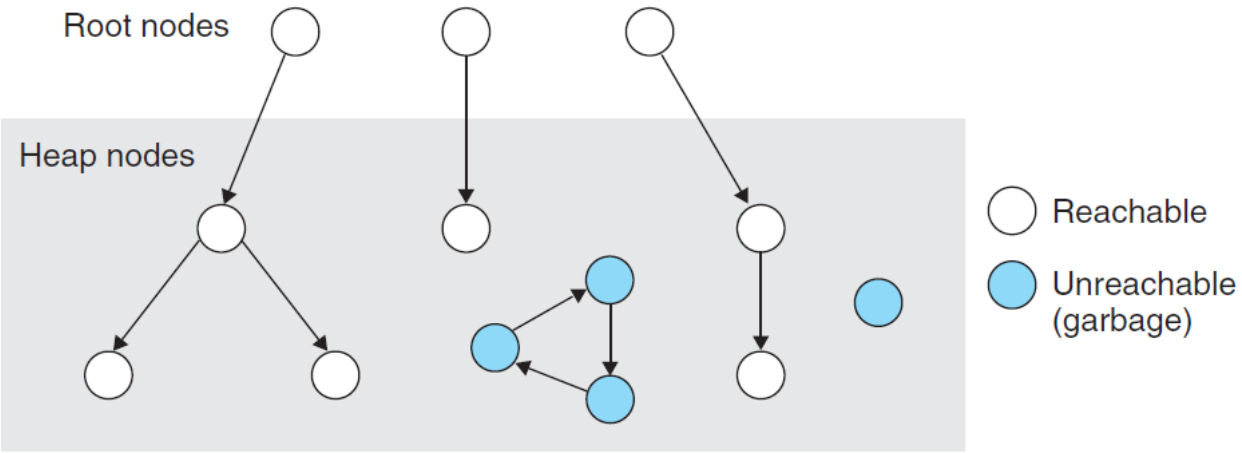



# ניהול אוטומטי של זיכרון דינמי עם Garbage Collector

- בלוקים של נתונים שתוכנית **לא יכולה** לגשת אליהם נחשבים garbage.
  - garbage collector מוצא את הבלוקים האלו ומחזיר אותם למאגר הבלוקים הפנויים.
  - garbage collector מופעל מידי פעם, בדרך כלל כאשר לא נותר מקום פנוי בערמה.
- אפשר לתאר את הנתונים שתוכנית **יכולה** לגשת אליהם כגרף עם קשתות מכוונות:

1. נתונים שהגישה אליהם היא ישירות ולא דרך מצביע - משתנים גלובליים ומשתנים במחסנית.
   נתונים אלו הם קבוצת השורש, מהם אפשר להגיע לערימה.
2. נתונים בערימה שאפשר לגשת אליהם דרך מצביעים (קשתות) מקבוצת השורש.
   וכן נתונים בערימה שאפשר לגשת אליהם רקורסיבית דרך אותם נתונים.

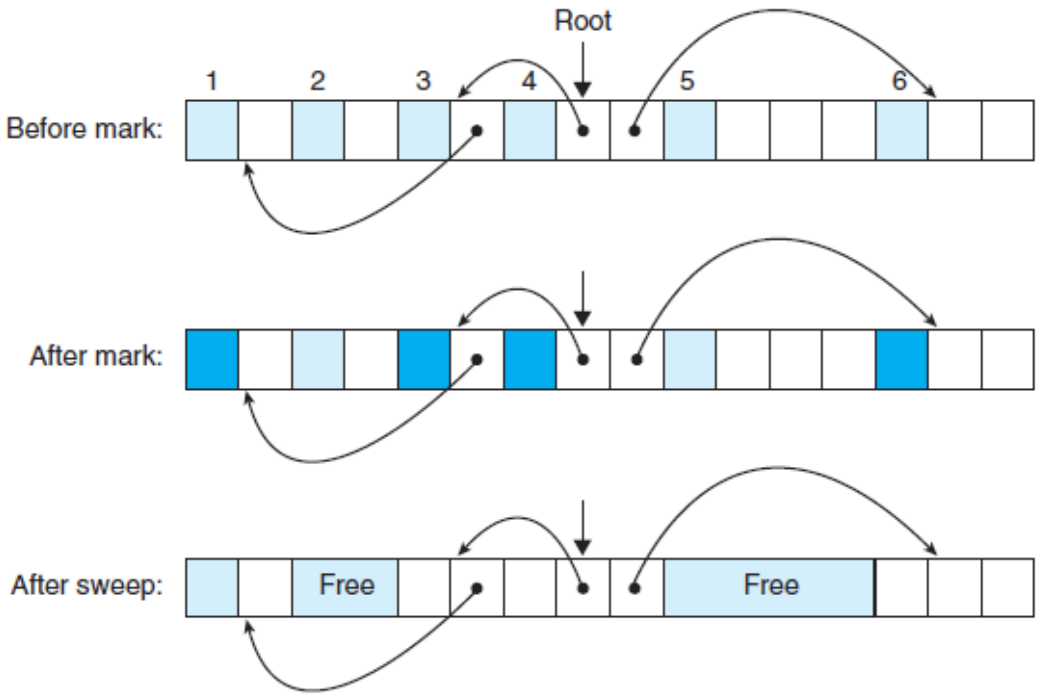
- בציור, העיגולים הם אובייקטים בזיכרון.
  - העיגולים הלבנים הם אובייקטים נגישים.
  - אובייקטים בערימה נגישים אם יש נתיב אליהם מקבוצת השורש.
  - העיגולים הכחולים אינם נגישים.



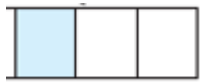
# Mark and Sweep

- אלגוריתם Mark and Sweep מוצא את הנתונים הלא נגישים (garbage) בשני שלבים:
- mark – סימון כל הבלוקים הנגישים בערימה:
  1. לפני הסימון, כל הבלוקים בערימה מסומנים כלא נגישים.
  2. סימון כל הבלוקים שאפשר להגיע אליהם מקבוצת השורש.
  3. קריאה רקורסיבית ל- mark עבור הבלוקים שהגענו אליהם.
  - אפשר להשתמש באחד מהביטים בכותרת הבלוק לצורך הסימון.
- sweep – שחרור כל הבלוקים שלא סומנו והחזרתם למאגר הבלוקים הפנויים.
- כדי שה- garbage collector יוכל למצוא את הנתונים הנגישים, הוא צריך לדעת איזה נתונים הם מצביעים.
  - בשפת Java אפשר בזמן ריצה לדעת איזה נתון הוא מצביע.
  - בשפת C ו- C++ אי אפשר לדעת, האפשרות היחידה היא להתייחס לכל מילה כאל מצביע.
    - ואז יש צורך לעבור על כל הנתונים של התכנית ולבדוק אם הם מצביעים לבלוק שהוקצה בערימה.
    - אם הם מצביעים לבלוק, לא מפנים אותו, למרות שיתכן שזה לא מצביע אלא מספר שבמקרה מצביע לבלוק.
    - יוצא שיתכן שנשאיר נתונים שהם garbage, אבל לא נפנה נתונים שאינם garbage.

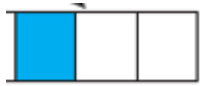
# Mark and Sweep




בציור, הריבועים הצבועים הם כותרת של בלוק, הלבנים הם נתונים של הבלוק.



כחול בהיר – כותרת בלוק שלא סומן



כחול – כותרת בלוק שסומן כנגיש

- פונקציית mark נקראת עבור כל מצביע לערימה מקבוצת השורש ונקראת רקורסיבית עבור הבלוקים שאפשר להגיע אליהם מאותו שורש.
  - בדוגמה, Root הוא מצביע לערימה, הוא מצביע לנתונים של בלוק 4.
  - בלוק 4 מכיל מצביע לבלוק 3 שמכיל מצביע לבלוק 1.
  - בלוק 4 מכיל עוד מצביע לבלוק 6.
  - כל הבלוקים שיש אליהם מצביע מסומנים כנגישים (בכחול).
  - הבלוקים הנותרים הם חופשיים.
- פונקציית sweep עוברת על כל הבלוקים בערמה ומחזירה את החופשיים למאגר הפנויים.

# Mark and Sweep

typedef void ***ptr;**

**ptr isPtr(ptr p):** If p points to an **allocated** block, returns a pointer to that block else NULL

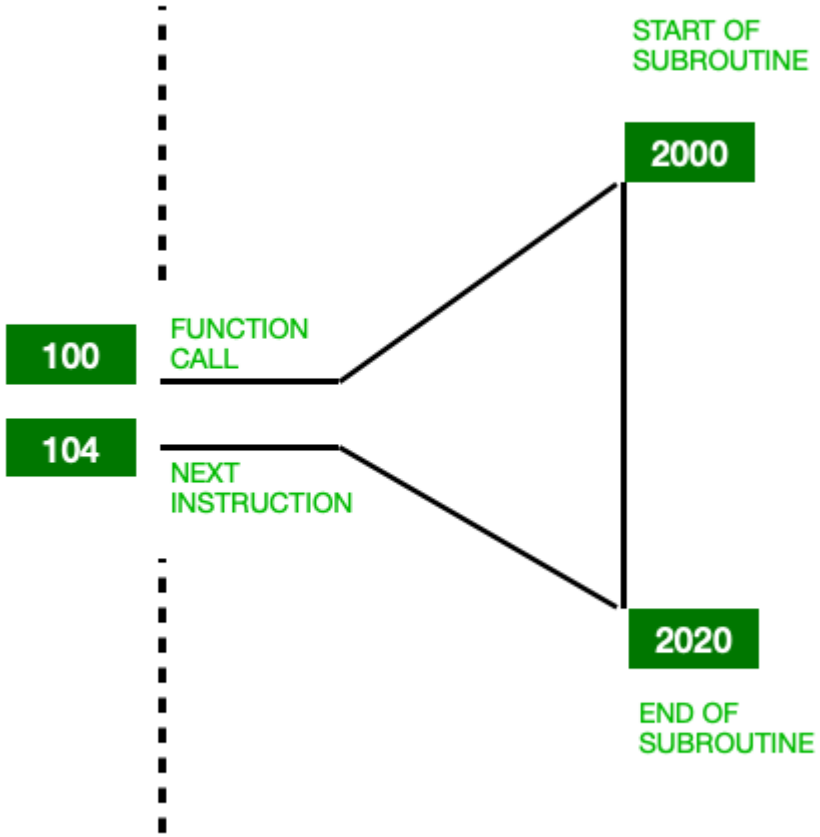
```
void mark(ptr p) {
   if ((b = isPtr(p)) == NULL)
     return;
   if (blockMarked(b))
     return;
   markBlock(b);
   len = length(b);
   for (i=0; i < len; i++)
     mark(b[i]);
   return;
}
```

```
void sweep(ptr b, ptr end) {
   while (b < end) {
      if (blockMarked(b))
         unmarkBlock(b);
      else if (blockAllocated(b))
         free(b);
      b = nextBlock(b);
   }
   return;
}
```

# נוהל קריאה לפונקציה

הצעדים הנדרשים בקריאה לפונקציה:

1. העברת פרמטרים לפונקציה הנקראת.
2. העברת הביצוע לפונקציה הנקראת.
3. הקצאת משתנים לפונקציה הנקראת.
4. ביצוע הפונקציה.
5. העברת התוצאה לפונקציה הקוראת.
6. החזרת הביצוע לפונקציה הקוראת.



# נוהל קריאה לפונקציה

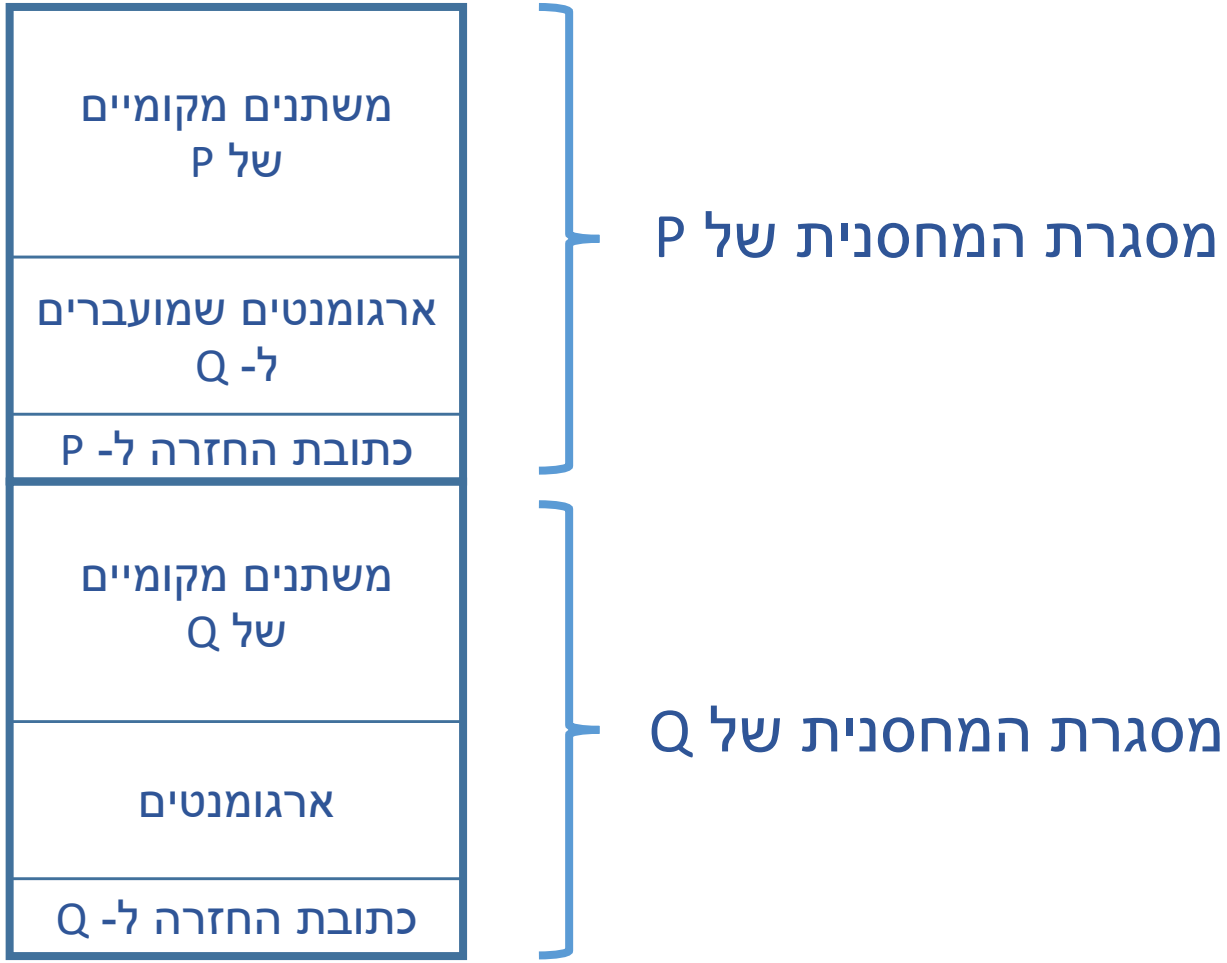
**מסגרת המחסנית** היא אזור במחסנית המשמש לניהול הקריאות לפונקציות.

1. כש- P קוראת ל- Q היא מבצעת קפיצה ל- Q.

   כדי ש- Q תוכל לחזור ל- P, **כתובת החזרה** (כתובת הפקודה שאחרי הקריאה) נשמרת במסגרת המחסנית.

2. אם P רוצה לקרוא ל-Q ולהעביר **ארגומנטים** ל- Q, היא שמה אותם במסגרת המחסנית.

3. אם הפונקציה P מגדירה משתנים מקומיים הם יהיו באוגרים או במסגרת המחסנית.



# המחסנית בזמן קריאה וחזרה מפונקציה